PROGRAMAÇÃO MODULAR

A Linguagem Java

Roberto
S.
Bigonha

PROGRAMAÇÃO MODULAR

A Linguagem Java

(SE 14)

Roberto S. Bigonha

Belo Horizonte, MG

29 de Junho de 2021

Roberto S. Bigonha: Pequisador Independente. PhD em Ciência da Computação pela Universidade da Califórnia, Los Angeles. Professor Titular Emérito da Universidade Federal Minas Gerais. Membro da Sociedade Brasileira de Computação. Áreas de interesse: Linguagens de Programação, Programação Modular, Estruturas de Dados, Compiladores, Semântica Formal.

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

B594	Bigonha, Roberto S. PROGRAMAÇÃO MODULAR: A Linguagem Java / Roberto da Silva Bigonha — Belo Horizonte, MG, 2021 Bibliografia. ISBN 978-65-00-22305-7 1. Linguagens de Programação 2. Linguagem Java I. Título. CDD: 005.1 CDU 004.43

Índice para catálogo sistemático:
1. Linguagens de Programação : Engenharia de Software

Sumário

Prefácio

A série de livros Programação Modular destina-se a alunos de cursos de graduação da área de Computação, como ciência da computação, análise de sistemas, matemática computacional, engenharia de computação e sistemas de informação. As técnicas apresentadas são voltadas para o desenvolvimento de programas de grande porte e complexos.

Este volume trata da apresentação da linguagem de programação Java, seus tipos básicos e sua estrutura de controle de execução, destacando seus recursos para programação modular. Parte-se do princípio de que sem os recurso linguísticos adequados para realizar as abstrações, que naturalmente surgem do projeto de programas, é muito difícil produzir sistemas verdadeiramente modulares. Para desenvolver algoritmos nem é preciso usar linguagens de programação, mas para programar sistemas de grande porte, condicionados por restrições de contorno reais, é indispensável que se tenha notação e formas adequadas para organizar os programas e implementar as abstrações, modelos e condições que surgem do processo de projeto. Sem os mecanismos e notação corretos para separar fisicamente interesses e implementá-los, o tamanho dos programas que podem ser desenvolvidos corretamente fica muito limitado, o controle de sua correção praticamente impossível, e o custo de manutenção proibitivo. A adoção de uma linguagem moderna como meio de comunicação e de implementação dos exemplos é, portanto, imperativo para dar clareza aos conceitos e ideias aqui defendidos. A linguagem Java serve a esses propósitos.

Agradecimentos

Agradeço aos diversos colegas que fizeram a leitura dos primeiros manuscritos, apontaram erros, indicaram as correções e também contribuíram com exemplos. Particularmente, agradeço a Mariza Andrade da Silva Bigonha o trabalho de revisão do texto, e a Luciano Capanema Silva, Thales Evandro Simas Júnior e Abdenago Alvim, alunos do Curso de Especialização em Informática da UFMG, a indicação de erros nas primeiras versões deste livro.

Roberto S. Bigonha

Capítulo 1

Estruturas Básicas

A linguagem Java é destinada à programação na *Web*, tendo sido projetada para ser independente de arquitetura e de plataforma de execução. Um programa em Java pode ser executado em qualquer plataforma disponível na rede mundial de computadores. Os tipos de dados básicos da linguagem possuem tamanho fixo e pré-definido e, normalmente, programas em Java não têm qualquer comprometimento com código nativo da plataforma em que são executados.

Programas em Java são compilados para *bytecode*, que é a linguagem de uma máquina virtual, a JVM, cujo interpretador está instalado nas principais plataformas de execução disponíveis no mercado, inclusive, como parte integrante dos principais navegadores da Internet.

Java é uma linguagem imperativa, orientada por objetos pura, com tipos seguros, recursos modernos para produção de módulos de boa qualidade e implementação de sistemas distribuídos na Web.

1.1 Identificadores

Identificadores em Java são usados para dar nomes a variáveis, rótulos, classes, pacotes, interfaces, enumerações, métodos, campos, parâmetros formais e constantes.

Todo identificador começa com uma letra, podendo ser seguido de uma sequência de qualquer tamanho contendo letras, dígitos, `$` ou o caractere de sublinhado (_).

As letras permitidas são as do conjunto Unicode (16 bits). O conjunto ASCII (8 bits) ocupa as primeiras posições do conjunto Unicode. Letras maiúsculas, minúsculas, acentuadas, não-acentuadas são distintas entre si na formação de identificadores.

Não há distinção na formação de identificadores em relação ao seu uso no programa, mas, com o objetivo de facilitar a legibilidade de programas, a seguinte padronização é recomendada e adotada neste texto:

- nomes de classes e de interfaces devem iniciar-se com letra maiúscula;
- nomes de constantes devem ser escritos com todas as letras em maiúsculo;
- nomes de outros elementos de um programa devem iniciar-se com letra minúscula.

Recomenda-se que se evite o uso do caractere "_"para separar as palavras de identificadores com mais de uma palavra. Nesses casos, é preferível apenas capitalizar a primeira letra das palavras internas do nome para facilitar a sua leitura.

Os seguintes identificadores são usados para designar palavras-chave da linguagem Java e, por isso, não podem ser usados para outros fins:

- Em declarações: `boolean, byte, char, double, final, float, int, long, private, protected, public, short, static, transient, void, volatile`
- Em expressões: `null, new, this, super, true, false`
- Em comandos de seleção: `if, else, switch, case, break, default`

- Em comandos de iteração: `for`, `continue`, `do`, `while`
- Em comandos de transferência de controle: `return`, `throw`, `try`, `catch`, `finally`
- Em classes: `abstract`, `class`, `instanceof`, `throws`, `native`, `interface`, `synchronized`
- Em módulos: `extends`, `implements`, `package`, `import`
- Reservadas para futuro uso: `cast`, `const`, `future`, `generic`, `goto`, `inner`, `operator`, `outer`, `rest`, `var`

1.2 Programas

Um programa em Java é uma coleção de classes, dentre as quais uma pode ser a principal. A classe principal é conceitualmente aquela pela qual a execução se inicia. Qualquer classe contendo o método `main`, de assinatura

```
public static void main(String[])
```

pode, em princípio, ser vista como principal. Por exemplo, a classe abaixo pode ser considerada principal, e sua execução, disparada da linha de comando.

```
1 public class Programa {
2    public static void main(String[] args) {
3       System.out.println("Olá!");
4    }
5 }
```

1.3 Tipos de dados

Os tipos de dados de Java podem ser primitivos ou referências. Os tipos primitivos são os tipos numéricos inteiros (`byte`, `char`, `short`, `int`, `long`), os numéricos fracionários (`float`, `double`) e

o booleano (`boolean`). Os tipos que denotam referências compreendem classes, interfaces e arranjos.

Variáveis de tipo primitivo obedecem ao modelo denominado **semântica de valor**, i.e., contêm diretamente um valor do tipo declarado. Variáveis declaradas do tipo referência seguem o modelo **semântica de referência**, e, assim, armazenam, não um valor do tipo declarado, mas o endereço do objeto que contém um valor do tipo declarado.

1.4 Tipo boolean

O tipo `boolean` é definido pelas constantes `true` e `false`, pela operação unária negação (`!`) e as operações binárias conjunção condicional (`&&`), disjunção condicional (`||`), conjunção (`&`), disjunção (`|`), disjunção exclusiva (`^`), atribuição (`=`), comparação por igual (`==`) e comparação por diferente (`!=`).

As operações binárias acima, ditas condicionais, operam em curto-circuito. Isto significa que essas operações somente avaliam seu segundo operando quando o resultado da avaliação do primeiro não for suficiente para deduzir o resultado da operação.

As demais operações avaliam sempre todos os seus operandos. Em uma linguagem como Java, que permite efeito colateral na avaliação de expressões, deve-se dar atenção a possibilidade de expressões serem ou não avaliadas.

O trecho de programa abaixo, que ilustra esse fato, atinge seu fim com `b == false` e `i == 9`:

```
boolean b, b1 = true;
int i = 10;
b = (b1 || (i++ == 10)) && (--i == 10);
```

1.5 Tipos numéricos

Os tipos numéricos inteiros são `byte`, `char`, `short`, `int` e `long`. Todos esses tipos possuem as operações de atribuição (`=`) e de comparação de valores, a saber: igual a (`==`), diferente de (`!=`), menor que (`<`), maior que (`>`), menor ou igual a (`=<`) e maior ou igual a (`>=`).

Os tipos numéricos `int`, `long`, `float` e `double` possuem ainda em comum as seguintes operações aritméticas: adição (`+`), subtração (`-`), multiplicação (`*`), divisão (`/`), módulo (`%`), pré ou pós-incremento (`++`), pré ou pós-decremento (`--`).

A operandos de tipo `int` e `long` aplicam-se também as operações de lógica bit-a-bit, que são conjunção (`&`), disjunção (`|`) e inversão (`~`), e as operações de deslocamento de bits: aritmético para a direita (`>>`), lógico para a direita (`>>>`) e lógico para a esquerda (`<<`). O deslocamento aritmético para a direita propaga o bit do sinal, enquanto o lógico preenche com zero os novos bits.

Os tipos numéricos com ponto flutuante são `float` de 32 bits, e `double` de 64 bits, representados de acordo com o padrão IEEE 754. As operações que definem cada um dos tipos são:

- aritméticas: adição(`+`), subtração(`-`), multiplicação(`*`), divisão(`/`), pré ou pós-incremento (`++`), pré ou pós-decremento (`--`)
- de atribuição: `=`
- de comparação: igual a(`==`), diferente de (`!=`), menor que(`<`), maior que (`>`), menor ou igual a (`=<`), maior ou igual a (`>=`)
- de deslocamento: para a esquerda(`<<`), aritmético para a direita (`>>`), lógico para a direita(`>>>`).

1.5.1 Tipo `int`

O tipo `int` compreende valores inteiros do intervalo

`[-2.147.483.648 , 2.147.483.647]`,

representados em 32 bits, formato complemento de 2.

As constantes desse tipo têm três formatos:

- decimal, que são sequências de algarismos decimais
- octal, que são sequências dos algarismos do conjunto de dígitos {`0,1,2,3,4,5,6,7`}, iniciadas pelo dígito `0` (zero)
- hexadecimal, que são iniciadas por `0x` (zero x) e formadas por dígitos hexadecimais ({`0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F`}).

1.5.2 Tipo `long`

O tipo `long` é o tipo dos inteiros representados em 64 bits, formato complemento de 2, e caracterizado pelas mesmas operações do tipo `int`. As suas constantes têm a mesma apresentação das constantes decimais do tipo `int`, porém devem ser acrescidas da letra `L` ou `l` (ele minúsculo) no fim e estão no intervalo

`[-9.223.372.036.854.775.808 , 9.223.372.036.854.775.807]`.

1.5.3 Tipo `byte`

O tipo `byte` compreende os valores inteiros no intervalo `[-128,127]`, representados em 8 bits, formato complemento de 2. O tipo não tem constantes e nem operações. Valores do tipo `byte` são convertidos para `int` para serem operados. Em atribuições a variáveis do tipo `byte`, conversões de tipo é necessária. O trecho de código abaixo mostra casos em que há necessidade de conversão dos resultados para ter os resultados em 8 bits:

```
int  x = 1;
byte b1 = 1, b2 = 2;
```

```
byte b3 = b1 + b2 ;          // Erro de compilação
byte b4 = (byte) (b1 + b2);
byte b5 = (byte) (512 + 3); // b5 recebe 3
byte b6 = x;                 // Erro de compilação
byte b7 = (byte) x;
```

1.5.4 Tipo `short`

Os inteiros do tipo **`short`** estão no intervalo `[-32.768, 32.767]`, sendo representados em 16 bits, formato complemento de 2. Análogo ao tipo **`byte`**, esse tipo também não possui constantes próprias nem operações. Há necessidade de conversão explícita para 16 bits nas atribuições. Valores do tipo **`short`** podem ser usados como operandos de operações do tipo **`int`**, sendo que esses operandos são convertidos para **`int`**.

O trecho de programa abaixo ilustra os usos permitidos e não permitidos de tipo **`short`**:

```
short b = 1, c = 2;
short d = b + c ;            // Erro de compilação
short e = (short) (b + c);
int x = b++;
short f = x;                 // Erro de compilação
short g = (short) x;
short h = b++;               // Erro de compilação
```

1.5.5 Tipo `char`

O tipo **`char`** é um tipo numérico caracterizado pelos inteiros no intervalo `[0,65.535]`, representados com 16 bits, sem sinal. Diferentente dos outros pequenos inteiros, **`byte`** e **`short`**, o tipo **`char`** possui constantes próprias, representadas como um caractere Unicode entre apóstrofes, em uma das seguintes formas:

- caractere individual, e.g., 'A', 'a'

- caracteres de escape:

`'\n'`	(nova linha)	`'\b'`	(retrocesso)
`'\"'`	(aspas)	`'\r'`	(retorno de carro)
`'\t'`	(tabulação)	`'\''`	(apóstrofe)
`'\f'`	(avanço de form.)	`'\\'`	(barra)

- sequência de escape octal: somente os oito bits menos significativos, isto é, `'\000'` a `'\377'`
- sequência Unicode: `'\uxxxx'`, onde `xxxx` representa 4 dígitos hexadecimais.

O trecho abaixo ilustra bons e maus usos do tipo `char`:

```
char x = 'a', y = 'z';
char w = x + y ;          // Erro de compilação
char t = (char) (x + y);
char u = (char) (x - y); // Erro de lógica?
```

O tipo `char` tem um comportamento dual, pois dependendo do contexto é tratado como numérico ou caractere, conforme mostra o programa abaixo.

```
1  public class Dual {
2     public static void main(String[] args) {
3        char x = 'A';
4        int y;
5        System.out.println("print(x)   = " + x);
6        System.out.println("print(y=x) = " + (y = x));
7        System.out.println("print(++x) = " + ++x);
8        System.out.println("print(y=x) = " + (y = x));
9        System.out.println("print(x)   = " + x);
10    }
11 }
```

que produz a esclarecedora saída:

```
print(x)   =  A
print(y=x) = 65
```

```
print(++x) =  B
print(y=x) = 66
print(x)   =  B
```

1.5.6 Tipo `float`

Valores de tipo `float` estão no intervalo `[-3.4E38, 3.4E38]` aproximadamente, com cerca de 6 dígitos significativos, armazenados em 32 bits, de acordo com padrão IEEE 754. As constantes têm o formato usual da notação científica, seguida de `f` ou `F`, e.g., `1E1F`, `1e1f`, `2.F`, `2f`, `.3f`, `3.14f`, `6.02E+23f`.

1.5.7 Tipo `double`

Valores de tipo `double` estão em `[-1.7E308, 1.7E308]` e têm cerca de 14 dígitos significativos, de 64 bits, representados de acordo com o padrão IEEE 754. As constantes têm o formato usual da notação científica, seguida ou não de `d` ou `D`, e.g., `1E1`, `1E1D`,`1e1d`, `2.`, `2d`, `.3D`, `3.14`, `6.02E+23`.

Promoções de `float` a `double` são automáticas. No sentido inverso, devem ser explícitas, conforme ilustrado pelo seguinte trecho de programa:

```
double x = '\n';            // atribui 10.0d a x
byte   b = (byte)x;
float  w = 6.5f;
float  y = 6.5;           // Errado: 6.5 é double
float  z = (float)6.5;
byte   b = 1;
float  t = b;
```

1.5.8 Conversões automáticas

Operações com inteiros nunca indicam ocorrências de estouro de valor para cima (*overflow*) nem para baixo (*underflow*), exceto

com divisão inteira (`/`) e resto de divisão inteira (`%`), em que a exceção `ArithmeticException` é levantada, quando o denominador for zero.

Inteiros menores, como `byte`, `char` e `short`, ocorrendo em uma expressão sem a presença de operandos `long`, são automaticamente promovidos a `int`, e a operação é realizada com 32 bits, produzindo resultado do tipo `int`. Por outro lado, numa operação de inteiros, em que pelo menos um dos operandos é do tipo `long`, os demais operandos são automaticamente promovidos a `long`, e a operação é executada usando precisão de 64 bits, produzindo resultado do tipo `long`.

A conversão de um numérico maior para um menor nunca é automática e requer indicação explícita da conversão, por meio do operador de *casting*, cujo uso é ilustrado por:

```
double d = 1.0E100D;
float  f = 1.0E20F;
long   t = 10000000000000000000;
int    i = 2147483647;
short  s = 20000;
char   c = 65535;
byte   b = 100;
double x = b;              // x recebe 100.0D
byte   y = (byte)  c;      // y recebe   -128
byte   n = (byte)  i;      // n recebe   -128
short  k = (short) i;      // k recebe -32768
char   m = (char)  i       // m recebe  65536
```

Observe que conversão explícita não verifica ocorrência de *overflow*, e que na conversão de `double` para `float` pode haver perda de precisão. A conversão de um valor negativo para `char` resulta em um valor positivo com a mesma configuração de bits.

1.6 Constantes simbólicas

Constantes são declaradas por meio de enumerações introduzidas por `enum`, ou então pelo modificador `final`, o qual informa ao compilador que a "variável" declarada não pode ter seu valor alterado após sua inicialização. O uso de `enum` para introduzir constantes simbólicas as coloca em um novo tipo distinto de qualquer outro no programa, haja vista que seu uso demanda qualificação.

No exemplo abaixo, a classe `Naipe1` e a enumeração `Naipe2` ilustram as duas formas de declaração de constantes:

```
1 class Naipe1 {
2   final static int PAUS    = 1;
3   final static int ESPADAS = 2;
4   final static int OUROS   = 3;
5   final static int COPAS   = 4;
6 }
7 enum Naipe2 {PAUS, ESPADAS, OUROS, COPAS}
```

Sobre uma constante definida em um `enum` somente pode-se aplicar operações de comparação por igual e por diferente, enquanto constantes definidas via `final` têm as operações do tipo em que foram declaradas, exceto atribuição, pois são finais:

```
1  public class UsoDeConstantes {
2    public static void main(String[] args) {
3       int x = Naipe1.OUROS + 1;
4       Naipe1.OUROS = 0;           // Erro
5       Naipe2 y = Naipe2.OUROS;
6       x = y++;                    // Erro
7       int z = Naipe2.OUROS + 1;   // Erro
8       Naipe2.OUROS = 0;           // Erro
9       .......
10   }
11 }
```

1.7 Expressões

Expressões em Java podem ser aritméticas, lógicas ou condicionais, e são formadas por operandos e operadores que podem ser unários, binários ou ternários. Pares de parênteses podem ser usados para dar clareza ou definir ordem de aplicação de operadores.

A expressão mais comum em Java é a expressão de atribuição, e.g., `a = b`, a qual retorna o valor da expressão `b`, e, colateralmente, atribui esse valor à variável `a`.

Operador	Operação
`-e`	troca sinal do operando `e` do tipo numérico
`!e`	nega logicamente o resultado de `e` do tipo `boolean`
`~e`	inteiro resultado da inversão dos bits do inteiro `e`
`++v`	valor de `v` após incrementar o seu conteúdo
`v++`	valor de `v` antes incrementar o seu conteúdo
`--v`	valor de `v` após decrementar o seu conteúdo
`v--`	valor de `v` antes de decrementar o seu conteúdo
`new t`	aloca objeto do tipo t e informa seu endereço
`(t)e`	converte, se possível, `e` para o tipo t

Tabela 1.1 Operadores unários

Outras expressões usam os operadores listados na Tabela 1.1 e Tabela 1.2, onde os termos `e`, `a`, `b` e `c` denotam expressões (operandos), `v`, uma variável na qual pode-se armazenar um valor, e `t` representa um tipo de dado.

Os operadores de deslocamento (`<<, >>, >>>`) e operadores de lógica bit-a-bit (`&, |, ^`) são aplicáveis a expressões do tipo inteiro, i.e., `int` ou `long`.

Na Tabela 1.2, os operandos `a` e `b` desses operadores são expressões do tipo `int` ou `long`, e `c` deve ser do tipo `int` ou de inteiro menor.

No deslocamento aritmético de bits para a direita, o bit do sinal é sempre replicado, de forma que o resultado tem o mesmo sinal do

operando, enquanto que no deslocamento lógico para a direita, o bit do sinal é substituído por zero, produzindo sempre um resultado positivo. No deslocamento para a esquerda, zeros são inseridos nos bits de menor ordem.

Java possui um operador de três operandos (`?:`), usado para definir expressões condicionais da forma: `e ? b : c`, onde `e` é uma expressão booleana, e os operandos `b` e `c`, expressões de um mesmo tipo. O valor da expressão acima é `b` ou `c`, conforme o valor de `e` seja verdadeiro ou falso, respectivamente. Por exemplo, a expressão `int x = (i>0 ? 10 : 20)` atribui o valor `10` a `x` se `i > 0`, ou valor `20`, caso contrário.

operação	Valor Produzido
`a & b`	*and* dos valores lógicos de `a` e `b`
`a | b`	*or* dos valores de `a` e `b`
`a ^ b`	*or* exclusivo dos valores de `a` e `b`
`a && b`	`false` se `a` for `false`, senão `b`
`a || b`	`true` se `a` for `true`, senão `b`
`a * b`	multiplicação dos numéricos `a` e `b`
`a / b`	divisão de `a` por `b`
`a % b`	resto da divisão do inteiro `a` pelo inteiro `b`
`a + b`	adição (numéricos) ou concatenação (strings) de `a` e `b`
`a - b`	subtração dos numéricos `a` e `b`
`a == b`	`true` se valor de `a` for igual ao valor de `b`
`a != b`	`true` se valor de `a` for diferente do valor de `b`
`a < b`	`true` se `a` for menor que `b`
`a > b`	`true` se `a` for maior que `b`
`a <= b`	`true` se `a` menor ou igual `b`
`a >= b`	`true` se `a` maior ou igual `b`
`a << c`	`a` com `c` bits deslocados para a esquerda
`a >> c`	`a` com `c` bits deslocados aritmeticamente p/ a direita
`a >>> c`	`a` com `c` bits deslocados logicamente p/ a direita
`a & b`	*and* bit a bit dos bits de `a` e `b`
`a | b`	*or* bit a bit dos bits de `a` e `b`
`a ^ b`	*or* exclusivo bit a bit dos bits de `a` e `b`

Tabela 1.2 Operadores binários

As prioridades dos operadores e as respectivas regras de associatividade estão definidas em ordem decrescente na Tabela 1.3. Note que algumas operações associam-se à esquerda, enquanto outras, à direita. Recomenda-se atenção à ordem de avaliação dos operandos, haja vista que expressões em Java podem ter efeito colateral.

Independentemente das prioridades dos operadores e da ordem de execução das operações fixadas pelo uso de parênteses, os operandos de uma expressão são normalmente avaliados da esquerda para a direita, e as operações são efetuadas somente após a avaliação de todos os operandos.

O exemplo abaixo, `OrdemDeAvaliacao1`, ilustra essa afirmação na avaliação do comando de atriuição da linha 9, que tem várias expressões com efeito colateral, o que obriga o entendimento da ordem definida na linguagem para inferir corretamente seu resultado.

Prioridade	Operadores	Associatividade
1	`~ ! ++ -- - (type)`	direita p/ esquerda
2	`* / %`	esquerda p/ direita
3	`+ -`	esquerda p/ direita
4	`<< >> >>>`	esquerda p/ direita
5	`< > <= >=`	esquerda p/ direita
6	`== !=`	esquerda p/ direita
7	`&`	esquerda p/ direita
8	`^`	esquerda p/ direita
9	`|`	esquerda p/ direita
10	`&&` (curto circuito)	esquerda p/ direita
11	`||` (curto circuito)	esquerda p/ direita
12	`?:`	direita p/ esquerda
13	`= += -= *= /= %= &=`	direita p/ esquerda
	`^= |= <<= >>>= >>=`	direita p/ esquerda

Tabela 1.3 Prioridade dos operadores

Observe os momentos em que operandos da expressão da linha 9 do programa `OrdemDeAvaliacao1` são avaliados e a ordem das operações pela análise do seguinte resultado por ele produzido:

```
v[0]=0 v[1]=1 v[2]=2 v[3]=3 v[4]=4 v[5]=5 v[6]=6 v[7]=7
Valor de i = 7
v[0]=6 v[1]=1 v[2]=2 v[3]=3 v[4]=4 v[5]=5 v[6]=6 v[7]=7
```

o qual mostra que o índice `i+=1` da linha 9 é avaliado em primeiro lugar, pois `v[0]` é o elemento atualizado pelo comando. A seguir, os índices dos usos do vetor `v` são avaliados. Em primeiro lugar, `i+=2` é executado, e o valor de `v[2]`, que neste instante contém `2`, é obtido. Em seguida, pela ordem, computam-se `i=3`, `v[3]`, `i+=4` e `v[7]`, produzindo um valor final `i=7`, o que se confirma pelo valor impresso, A primeira operação aritmética feita é o cálculo de `v[3]-v[7]`, que produz `-4`. Depois faz-se a operação `2-(-4)` para produzir o valor `6`, que é atribuído a `v[0]`.

```
1  public class OrdemDeAvaliacao1 {
2    public static void main(String[] args) {
3      int i = -1;
4      int[] v = new int[8];
5      for (int k=0; k < v.length; k++) {
6        v[k] = k;
7        System.out.print("v[" +k+ "]=" + v[k] + " ");
8      }
9      v[i+=1] = v[i+=2] - (v[i=3] - v[i+=4]);
10     System.out.println("\nValor de i = " + i);
11     for (int k=0; k < v.length; k++)
12       System.out.print("v[" +k+ "]=" + v[k] + " ");
13   }
14 }
```

Os operandos de *and* (`&&`), *or* (`||`) e ternário (`?:`) são avaliados somente à medida que sejam necessários, e abortam a avaliação da expressão tão logo seja possível inferir o seu resultado. Como a avaliação de expressões pode ter efeito colateral, cuidado é recomendado no uso desse tipo de expressão.

```
1  public class OrdemDeAvaliacao2 {
2    public static void main(String[] args) {
3      int i = -1;
4      int[] v = new int[8];
5      for (int k=0; k < v.length; k++) {
6        v[k] = k;
7        System.out.print("v[" + k + "]=" + v[k] + " ");
8      }
9      v[i+=1] = ((i==10) ? v[i+=2] : v[i+=1] - v[i+=4]);
10     System.out.println("\nValor de i = " + i);
11     for (int k=0; k < v.length; k++)
12       System.out.print("v[" +k+ "]=" +v[k]+ " ");
13   }
14 }
```

No programa `OrdemDeAvaliacao2`, o índice `i=+2` do elemento `v[i+=2]` da linha 9 não é computado, e o programa produz a saída:

```
v[0]=0 v[1]=1 v[2]=2 v[3]=3 v[4]=4 v[5]=5 v[6]=6 v[7]=7
Valor de i = 5
v[0]=-4 v[1]=1 v[2]=2 v[3]=3 v[4]=4 v[5]=5 v[6]=6 v[7]=7
```

1.8 Arranjos

Arranjo, também chamado de vetor, é uma estrutura de dados linear unidimensional composta de zero ou mais elementos de um mesmo tipo. Arranjos são objetos criados dinamicamente, e aos quais se podem ter acesso por meio da indexação de sua referência por um inteiro do tipo `int`.

Se `v` é uma referência a um arranjo, então `v[i]` é uma referência ao seu elemento de ordem `i+1`, desde que o valor de `i` seja de um inteiro no intervalo $[0, \texttt{v.length-1}]$.

O tamanho do arranjo alocado é definido no momento de sua criação. Índices de arranjos são do tipo `int`. Valores de inteiros de tipos menores são automaticamente promovidos a `int`. Não se

pode indexar arranjos por inteiros do tipo `long`. Indexação fora de limites gera a exceção `IndexOutOfBoundsException`.

Em uma referência a um arranjo, a expressão anterior aos colchetes `[]` é avaliada antes de qualquer índice e, na alocação de um arranjo, as dimensões são avaliadas, uma por uma, da esquerda para a direita.

Objetos declarados do tipo arranjo são alocados no **heap** via operador `new`, o qual retorna o endereço do objeto alocado. Os objetos alocados no **heap** são liberados automaticamente quando não forem mais necessários.

Na criação de um objeto arranjo, deve-se especificar o tipo de seus elementos, o número de níveis de aninhamento de arranjos e, pelo menos, o tamanho da primeira dimensão. As dimensões omitidas são sempre as que ficam mais à direita da última especificada, isto é, se omite-se a i-ésima dimensão, todas as dimensões de ordem maior que i também devem ser omitidas.

No momento da alocação, todos os elementos de um objeto arranjo são automaticamente iniciados conforme o tipo de seus elementos. Elementos numéricos recebem valor inicial zero, booleanos, `false`, e referências são iniciadas com o valor `null`. A declaração `T[] v = new T[MAX]`, onde `T` é um nome de um tipo válido em Java, e `MAX` é uma expressão cuja avaliação produz um valor do tipo `int`, tem o efeito de:

- definir `v` como uma referência para o arranjo
- alocar um objeto arranjo de tamanho `MAX`
- iniciar cada elemento do arranjo conforme o tipo `T`
- atribuir o endereço da área alocada à referência `v`.

A declaração `int a [] = new int[9]` produz a Fig. 1.1. E as declarações `int[] a = new int[9]` e `int[] b = new int[8]` produzem a Fig. 1.2.

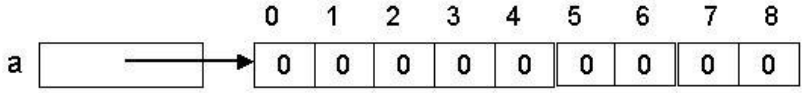


Figura 1.1 Alocação de arranjos I

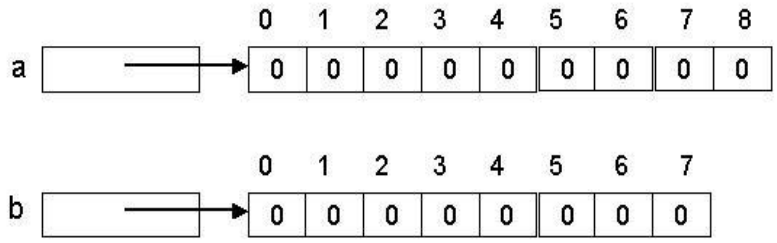


Figura 1.2 Alocação de arranjos II

A sequência de comandos `a[0]=1; a[1]=2; a[2]=3` atribui novos valores às posições de índices 0, 1 e 2, respectivamente, do arranjo `a`, resultando na configuração descrita pela Fig. 1.3.

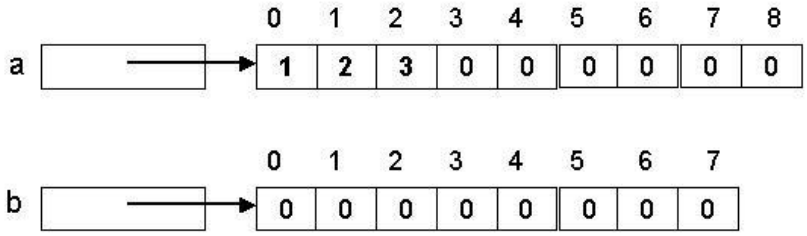


Figura 1.3 Alocação de arranjos III

O comando `b = new int[5]` aloca um novo objeto arranjo de tamanho suficiente para armazenar exatamente 5 inteiros e atribui seu endereço na mesma variável `b`, alterando a alocação descrita na Fig. 1.3 para a da Fig. 1.4.

Na Fig. 1.4, a área anteriormente apontada por `b` ficou sem uso e pode ser recolhida pelo coletor de lixo, se não houver no programa outro apontador para essa área. Área alocada para qualquer objeto

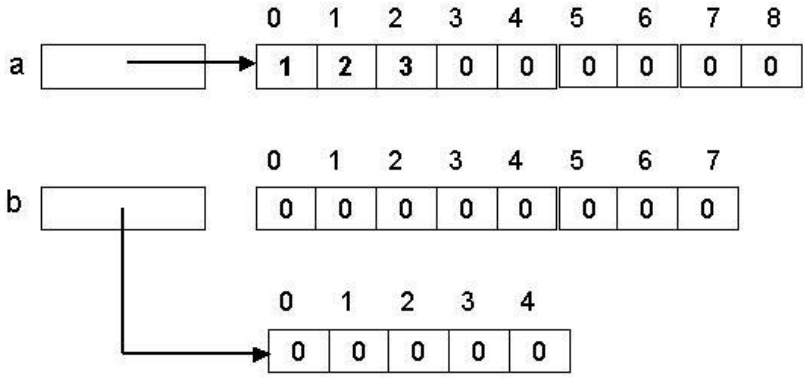


Figura 1.4 Alocação de arranjos IV

é liberada automaticamente quando esse objeto não for mais referenciado no programa e poderá ser reusada se houver necessidade de memória para continuar sua execução.

Arranjos multidimensionais podem ser criados por meio da declaração de arranjos aninhados ou arranjo de arranjos, como mostra o exemplo abaixo, no qual `a` é declarado para apontar para arranjo de arranjos de inteiros:

```
int [][] a;
a = new int[4][5];
```

Essa sequência de comandos declara a referência `a`, aloca um vetor de 4 referências, sendo cada uma para arranjos de `int` e inicia `a` com o endereço desse vetor. A seguir, alocam-se 4 vetores, todos zerados, de 5 inteiros cada um, e inicia cada posição do vetor apontado por `a` com o endereço de um desses vetores de inteiros. O resultado é o equivalente a uma matriz $4X5$, conforme ilustrado pela Fig. 1.5.

A alocação de arranjo de arranjos pode ser parcial, isto é, pode-se omitir, no operando de `new`, a dimensão de arranjos da direita para a esquerda, exceto a do primeiro índice.

Considere a declaração `int [][] b = new int[6][]` que produz a alocação exibida na Fig. 1.6, onde somente um arranjo de

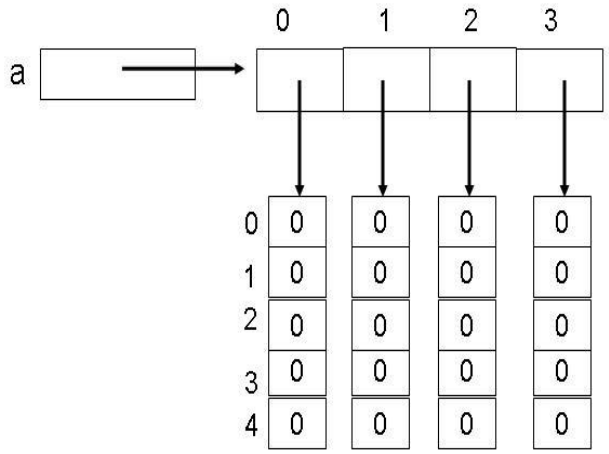


Figura 1.5 Alocação de arranjos V

referências zeradas para vetores de inteiros é alocado. As seis referências criadas podem ser iniciadas posteriormente.

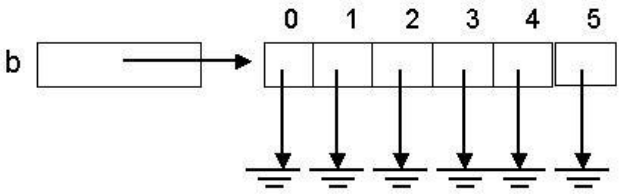


Figura 1.6 Alocação de arranjos VI

Se, a seguir, os comandos

```
b[0] = new int[5];
b[3] = new int[3];
```

forem executados tem-se uma *matriz* cujas colunas são desiguais, conforme mostra a Fig. 1.7.

Em declarações contendo uma única referência, a posição dos colchetes em relação ao elemento declarado não faz diferença. As declarações `char c[][]`, `char[]c[]` e `char[][]c` são equivalentes. Todavia, quando há mais de uma referência declarada em uma mesma construção, colchetes podem ser colocados ou junto ao tipo ou junto a cada um dos elementos declarados, como em ilustrado em `short[] i[], j, k[][]`.

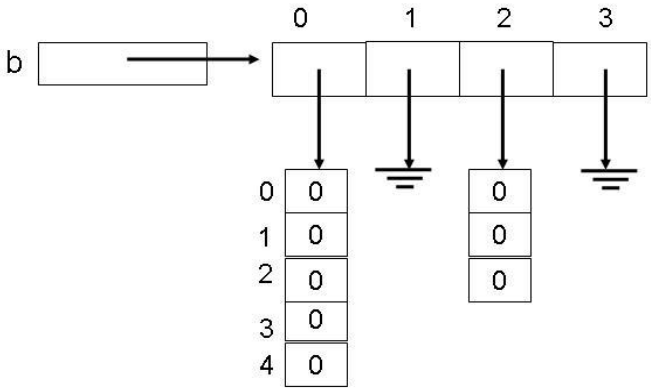


Figura 1.7 Alocação de arranjos VII

A regra da linguagem Java estabelece que colchetes (`[]`) colocados junto ao tipo aplicam-se a todos os elementos declarados, enquanto aqueles colocados junto a cada uma das referências declaradas aplicam-se somente à respectiva referência.

No exemplo acima, `i[]`, `j` e `k[][]` são referências para arranjo de `short`. Por conseguinte, `i` é uma referência para um arranjo de arranjos de `short`, `k` é uma referência para arranjo de arranjos de arranjos de `short`.

Arranjos podem ser iniciados no momento de sua declaração. O valor de iniciação do arranjo pode ser definido por uma expressão que consiste em uma lista, colocada dentro de chaves ({ }), de expressões separadas por vírgulas. O número de expressões na lista indica o comprimento do arranjo, e cada expressão que especifica o valor do respectivo elemento do arranjo deve ter tipo *compatível para atribuição* com o declarado para os elementos do arranjo. O operador de alocação de objeto `new` pode ser omitido. Assim a declaração

```
int[] a = {10, 20, 30, 40};
```

é equivalente a

```
int[] a = new int[]{10, 20, 30, 40};
```

e produz o efeito indicado pela Fig. 1.8.

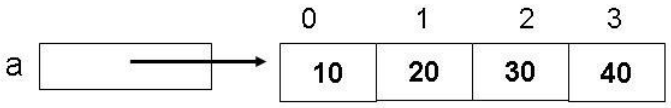


Figura 1.8 Alocação de arranjos VIII

No caso de arranjos multidimensionais, deve-se codificar um iniciador de arranjo dentro de cada elemento inicializado da dimensão anterior. Se houver mais de duas dimensões, o processo deve se repetir. Um exemplo de alocação e iniciação de um arranjo bidimensional é ilustrado pela declaração:

```
int[][] a = {{1,2,3,4,5},{6,7},{8,9,10},{11,12,13,14,15}};
```

que produz o resultado representado pela Fig. 1.9.

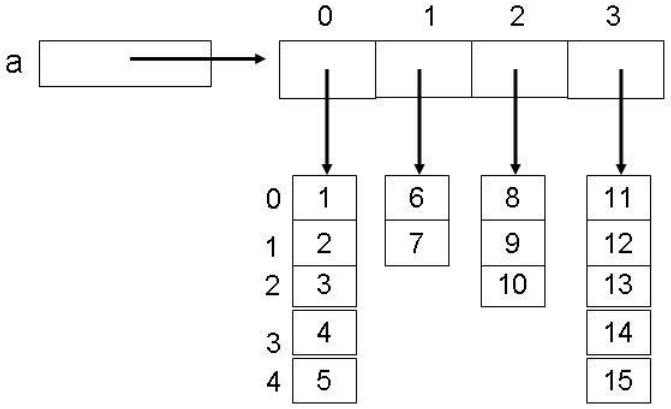


Figura 1.9 Alocação de Arranjo IX

A operação `a.length` informa o número de elementos alocados para o objeto arranjo diretamente apontado pela referência `a`, considerado apenas a primeira dimensão.

Os elementos de arranjos apontados pelos elementos do vetor referenciado por `a` não são computados no tamanho `a.length`. O tamanho do arranjo apontado por `a[0]` é dado por `a[0].length`, o do arranjo apontado por `a[1]`, por `a[1].length`, etc, como ilustra o programa `Tamanhos` abaixo,

```
1  public class Tamanhos {
2   public static void main(String [] args) {
3    int [][] a = new int[4][];
4    int [][] b = new int[4][5];
5    a[0] = new int[1];  a[1] = new int[2];
6    a[2] = new int[3];  a[3] = new int[4];
7    System.out.print("a[0]:" + a[0].length);
8    System.out.print(", a[1]:" + a[1].length);
9    System.out.print(", a[2]:" + a[2].length);
10   System.out.print(", a[3]:" + a[3].length);
11   System.out.print(", a:" + a.length + ", b:" + b.length);
12  }
13 }
```

que produz a saída: `a[0]:1, a[1]:2, a[2]:3, a[3]:4, a:4, b:4`

1.9 Comandos

Os tipos de comandos de Java são: atribuição, bloco, condicional (**`if`** e **`switch`**), repetição (**`while`**, **`do-while`**, **`for`**, **`for`** aprimorado), chamada de método, retorno de método (**`return`**), controle de repetição (**`continue`** e **`break`**), tratamento de exceções (**`try-catch- finally`**) e controle de acesso a regiões críticas (**`synchronized`**).

1.9.1 Atribuição

Comandos de atribuição são de fato uma expressão com o operador binário `=`, cujo valor é o do seu segundo operando, e que, como efeito colateral, atribui o valor de seu segundo operando à posição de memória especificada pelo primeiro operando.

Comandos de atribuição da forma **`x op= e`**, onde **`x`** é um termo que denota uma variável, **`op`**, um dos operadores binários de Java,

e `e` é uma expressão, são equivalentes a avaliar o termo `x` para produzir um endereço de memória, obter o valor associado a esse endereço, avaliar a expressão `e`, executar a operação `op` entre o valor associado a `x` e o valor da expressão `e`. O resultado obtido é armazenado no endereço denotado por `x`.

Note que devido à possibilidade de haver efeito colateral na avaliação da expressão `x`, a forma `x op= e` nem sempre é equivalente a `x = x op e`. Por exemplo, o comando de atribuição

```
x[i++] = x[i++] + 1
```

não é o mesmo que o comando

```
x[i++] += 1,
```

porque os valores de `i` após a execução dos comandos seriam diferentes e também assim seria a posição do arranjo atualizada.

1.9.2 Bloco

Bloco é uma estrutura que permite reunir um grupo de comandos e definir um escopo local de declarações. Um comando bloco tem um dos formatos:

- { *declarações e comandos* }
- { *comandos* }

Variáveis declaradas sem inicializador em um bloco não recebem valor inicial e devem ser explicitamente inicializadas antes de seu uso. Isso é verificado pelo compilador Java, que rejeita programas em que a inicialização das variáveis do bloco não puder ser verificada e garantida.

Bloco deve ser usado sempre que se quiser codificar vários comandos onde apenas um for permitido pelas regras sintáticas ou se desejar delimitar o escopo de certas declarações.

Por razões de legibilidade, Java não permite a ocultação de nomes em escopo aninhados, como tentou-se fazer com `j` na função

`main` do programa abaixo:

```
1 public static void main(String[] args) {
2    int i = 10, j ;
3    if (i == 10) {
4       int j = 100;  // Erro de compilação
5       i += j;
6    }
7    j = i*10;
8 }
```

Para resolver esse problema há duas soluções: mudar o nome de uma das variáveis em conflito ou trocar a declaração mais externa de lugar.

A troca de uma declaração de lugar pode resolver o problema porque o escopo de uma declaração em Java inicia-se somente no ponto em que ocorre, estendendo-se até o fim do bloco, como pode-se ver no trecho de programa a seguir.

```
1 public static void main(String[] args) {
2    int i = 10;
3    if (i == 10) { int j = 100; i += 10;}
4    int j;
5    j = i*10;
6 }
```

1.9.3 Comando `if`

O comando `if`, que permite escolher entre duas possibilidades de continuação na execução do programa, tem um dos formatos:

- `if (` *expressão_booleana* `)` *comando*
- `if (` *expressão_booleana* `)` *comando1* `else` *comando2*

O uso do comando `if` é exemplificado no programa abaixo, que recebe como parâmetro o *string* fornecido na linha de comando e inspeciona seu quarto caractere:

```
1  public class UsoDoIf {
2     public static void main(String args[]) {
3        String s = arg[0];
4        System.out.print("Quarto caractere de arg[0] ");
5        if (s.length > 3) {
6           char c = s.charAt(3);
7           if (c == 'A')
8              System.out.print("é um A");
9           else System.out.print("não é um A");
10       } else System.out.print("é indefinido");
11    }
12 }
```

Se a ativação do programa `UsoDoIf` for

```
C:>java UsoDoIf M1gA---2kdjgadsdwHH88
```

o texto impresso será:

```
Quarto caractere de arg[0] é um A
```

1.9.4 Comando `switch`

O comando **`switch`**, que permite fazer uma seleção de um fluxo de controle dentre um conjunto de possibilidades pela comparação de um valor inteiro com uma lista de constantes do mesmo tipo, tem o seguinte formato geral:

```
switch ( expressão-cabeçalho ) {
   case expressão-constante1:
        ...
        break;
   case expressão-constante2:
        ...
        break;
   ....
   default:
        ...
}
```

A expressão do cabeçalho do `switch` deve produzir valores do tipo `int`, `short`, `char`, `byte` ou do tipo enumeração. As expressões constantes dos casos dos `switch` são expressões cujos valores são computáveis em tempo de compilação.

A cláusula `default` é opcional e pode ser colocada em qualquer posição em relação aos casos do `switch`.

A semântica do comando `switch` compreende avaliar a expressão do tipo inteiro do seu cabeçalho e compará-la, sequencialmente, de cima para baixo no código, com os valores das expressões-constante que encabeçam cada uma das cláusulas `case` especificadas.

O controle da execução é passado para os comandos associados à primeira constante que satisfizer à comparação. A partir daí, todos os comandos que se encontram textualmente abaixo, dentro do `switch`, são executados normalmente, inclusive os dos casos que se seguem.

Para impedir que o fluxo avance sobre os comandos das cláusulas seguintes à escolhida, o comando `break`, definido a seguir, deve ser usado apropriadamente. Após o `switch`, o fluxo segue normalmente.

Se nenhuma comparação com expressões dos `cases` tiver sucesso, os comandos da cláusula `default` ganham o controle da execução, e depois o fluxo segue normalmente após o switch. Entretanto, se não houver cláusula `default` especificada no comando, a execução simplesmente continua no comando que segue o comando `switch`.

Como os casos são testados sequencialmente, o custo de execução de um `switch` pode ser alto quando houver muitos `case`s, e, nesses casos, outras soluções poderiam ser preferíveis.

O programa `Switch` abaixo imprime `"x = 3"`. E veja que o fluxo de execução executa os comandos associados aos casos 2, 3

e 4 antes de deixar o `switch` pela execução do primeiro `break` encontrado.

```
1  public class Switch {
2     public static void main(String[] args) {
3        int x = 0, k = 2;
4        switch (k) {
5           case 1:  x = x + 1;
6                    break;
7           case 2:  x = x + 1;
8           case 3:  x = x + 1;
9           case 4:  x = x + 1;
10                   break;
11          case 5:  x = 5;
12          default: x = x + 1000;
13       }
14       System.out.println("x = " + x);
15    }
16 }
```

1.9.5 Comando `while`

O comando de repetição `while` tem uma das formas:

- `while` (*expressão-booleana*) *corpo*
- `do` *corpo* `while` (*expressão-booleana*)

Na primeira forma, a *expressão-booleana* é avaliada e, se resultar `true`, o comando *corpo* é executado. Após o término da execução do *corpo*, a *expressão-booleana* é reavaliada e se ainda continuar `true`, o corpo é executado novamente. Essa repetição termina quando a avaliação da *expressão-booleana* produzir `false`, momento em que o fluxo de execução segue após o comando `while`. A execução também pode terminar com a execução de um `break` diretamente colocado no corpo do comando.

O programa abaixo tem o efeito de imprimir os valores armazenados no vetor `a`, iniciando pelo primeiro valor diferente de zero, e sua saída é `3 5 10 1 2 0 4`. Observe que se o arranjo `a` estivesse todo zerado, nada seria impresso, porque `h` teria um valor acima do último índice válido.

```
1  public class While {
2     public static void main(String[] arg) {
3        int[] a = {0,0,0,0,0,3,5,10,1,2,0,4};
4        int h = 0;
5        while ( h < a.length && a[h] == 0 ) h++;
6        while ( h < a.length ) {
7           System.out.println(a[h] + " ");
8           h++;
9        }
10    }
11 }
```

Na segunda forma do comando `while`, o chamado `do-while`, o teste da condição de parada é feito após cada execução do corpo do comando. Nesse caso, é garantida a execução do corpo do comando pelo menos uma vez, como ilustra o seguinte programa:

```
1  public class DoWhile {
2     public static void main(String[] arg) {
3        int[] a = {0,0,0,0,0,3,5,10,1,2,0,4};
4        int h = 0;
5        do {h++;} while (h < a.length && a[h-1] == 0);
6        if ( h < a.length )
7           do {
8               System.out.println(a[ht]);
9               h++;
10          } while ( h < a.length );
11    }
12 }
```

O programa `DoWhile` produz o mesmo resultado do programa `While`, embora pareça um pouco mais complicado. Isso ilustra que a correta escolha do tipo de comando `while` é importante.

1.9.6 Comando `for`

O comando de repetição `for` tem a forma:

- `for (` *iniciação* `;` *condição* `;` *avanço* `)` *corpo*

A execução desse tipo de comando inicia-se com a avaliação da expressão *iniciação*. A seguir, a expressão *condição* é avaliada, e se resultar `true`, o comando *corpo* é executado, e supondo que o corpo não contenha diretamente um comando `breaK`, após o término de sua execução, a expressão *avanço* é avaliada, e em seguida reavalia-se a expressão *condição* do comando `for`, e se ainda continuar produzido o valor `true`, o comando *corpo* é re-executado, a expressão *avanço*, reavaliada, e a *condição*, novamente testada.

O processo de repetição somente encerra-se quando a expressão *condição* do `for`, após a avaliação de *avanço*, resultar em `false`, momento em que a execucão prossegue com o próximo comando, ou então quando ocorrer a execução de um comando `break` encontrado diretamente na sequência de execução do corpo.

A expressão de iniciação do comando `for` pode conter declaração de variáveis. O escopo de uma variável declarada na expressão de iniciação de um comando `for` restringe-se ao próprio comando. Ressalva-se que somente uma declaração de variáveis pode ser especificado em cada `for`. Isso não é problema, pois uma declaração pode introduzir mais de uma variável. Por exemplo, o seguinte comando for, com duas variáveis de controle inteiras `i` e `j`, é válido:

```
for (int i=0, j=a.length-1 ; i<a.length ; i++,j--) {
  System.out.println("a["+i+"]+a["+j+"]= " + (a[i]+a[j]));
}
```

Por outro lado, o `for` abaixo não é válido, porque há mais de uma declaração na expressão de *iniciação*.

```
for (int i=0, int j=a.length-1 ; i<a.length ; i++,j--) {
  System.out.println("a["+i+"]+a["+j+"]= " + (a[i]+a[j]));
}
```

O programa abaixo ilustra o uso de variáveis de `for` locais.

```
1  public class VarDeControle {
2    public static void main(String[] args) {
3      int i;
4      int s = 0;
5      int a [] = {1,2,3,4,5,6,7,8,9};
6      for (i = 0; i < a.length ; i++) s += a[i];
7      System.out.println("Soma de a = " + s);
8      for (int j = a.length - 1, j >= 0; j--) {
9        System.out.println(""a[" + j + "]= " + a[j]);
10     }
11   }
12 }
```

No programa abaixo há dois tipos de erros: (i) há duas declarações na expressão *iniciação* dos dois comandos `for`; (ii) há declarações que indevidamente ocultam as variáveis `i` e `j` declaradas na linha 4.

```
1  public class ForErrado {
2    public static void main(String [] args) {
3      int a [] = {1,2,3,4,5,6,7,8,9};
4      int i = 9, j = 9;
5      for (int i=0, int j=a.length-1; i<a.length; i++,j--)
6        System.out.println("a[" + i + "] + a[" + j + "]= "
7                                      + (a[i] + a[j]));
8      for (int i=0, int j=a.length-1; i<a.length; i++,j--)
9        System.out.println("a[" + i + "]+a[" + j + "]= "
10                                     + (a[i] + a[j]));
11   }
12 }
```

1.9.7 Comando `for` aprimorado

O comando **`for`** aprimorado permite o acesso sequencial de leitura aos elementos de objeto contêiner[1], que pode ser um arranjo ou uma coleção, a qual é um objeto de algum tipo da hierarquia da classe **`Collection`**. O formato de um **`for`** aprimorado é:

- **`for`** **`(`** *tipo variável-de-controle* **`:`** *contêiner* **`)`** *comando*

A *variável-de-controle* do **`for`** deve ser de tipo compatível com o dos elementos do contêiner. A cada iteração, a *variável-de-controle* denota um dos elementos do contêiner, na ordem por ele definida.

Não é permitido atribuir valores à *variável-de-controle*, pois ela deve ser usada apenas para acesso aos elementos da contêiner e não pode ser usada para modificá-los.

```
1 public class ForAprimorado {
2    public static void main(String[] args) {
3       int s = 0;
4       int [] a = {1,2,3,4,5,6,7,8,9};
5       for (int x : a) s += x;
6       System.out.print(s);
7    }
8 }
```

O programa acima ilustra o uso do **`for`** aprimorado, calculando a soma dos elementos de um vetor **`a`** de inteiros.

1.9.8 Rótulos de comandos

Comandos podem ser rolutados para que comandos **`break`** ou **`continue`** façam a eles referências. O rótulo é um identificador que, seguido de **`:`**, é colocado antes de um comando.

[1] Um contêiner é um objeto que contém uma coleção de outros objetos de um dado tipo.

1.9.9 Comando `continue`

Comando **`continue`** somente pode ocorrer dentro de comandos **`for`**, **`while`** ou **`do-while`**. Em **`while`** ou **`do-while`**, **`continue`** causa a expressão booleana de controle de iterações do comando de repetição ser novamente avaliada e testada para decidir quanto à execução da próxima iteração. Dentro de **`for`**, **`continue`** causa a expressão de *avanço* ser reavaliada e a *condição* do comando de repetição ser novamente testada para decidir quanto à execução da próxima iteração, i.e., o comando **`continue`** causa o avanço da repetição para a próxima iteração. Seus formatos possíveis são:

- **`continue`**, que se refere ao comando de repetição envolvente mais próximo
- **`continue`** *rótulo*, que se refere ao comando de repetição envolvente que tem o *rótulo* especificado no comando.

O programa abaixo conta os números não-primos no intervalo `[3,10.000.000]`. O comando de repetição mais interno é interrompido quando um divisor do número em análise for encontrado.

```
1  public class Continue1 {
2    public static void main(String[] args) {
3      int i = 3, j, k = 0;
4      L: while (i < 10000000) {
5           j = 2;
6           while(j < i) {
7              if (i % j ==  0) {k++; i++;continue L; }
8              j++;
9           }
10          i++;
11     }
12     System.out.print("Não-Primos = " + k);
13   }
14 }
```

1.9.10 Comando `break`

O comando **`break`** somente pode ocorrer dentro de comandos **`for`**, **`switch`**, **`while`** ou **`do-while`**. Ele se apresenta em dois formatos:

- **`break`**, que termina o comando **`switch`**, **`for`**, **`while`** ou **`do`** envolvente mais próximo
- **`break`** *rótulo*, que termina o comando **`switch`**, **`for`**, **`while`** ou **`do-while`** envolvente que tem o rótulo indicado

O programa **`Break1`** abaixo também conta todos os números que não são primos no intervalo `[3, 10.000.000]`. O comando de repetição mais interno é interrompido assim que um divisor for encontrado, evitando-se completar todas as iterações.

```
1  public class Break1 {
2     public static void main(String[] args) {
3        int i = 3, j, k;
4        L: while (i < 1000000) {
5              j = 2;
6              while(j < i) {
7                 if (i % j ==  0) { k++; break; }
8                 j++;
9              }
10             i++;
11       }
12       System.out.print("Não-Primos = " + k);
13    }
14 }
```

Um comando **`break`** sem rótulo que ocorre diretamentre dentro de um **`switch`** tem efeito somente nesse comando, mesmo se o **`switch`** estiver dentro de um **`for`** ou **`while`**. Caso deseje-se abandonar loop envolventes, deve-se rotular os comandos a que o **`break`** se refere, como mostra o programa abaixo, onde há **`break`** de **`switch`** e de **`for`** no mesmo contexto.

```
1  public class Break2 {
2     public static void main(String[] args) {
3        int a = 0;
4        L: for (int j = 0; j < 6; j++) {
5            switch (j) {
6               case 0 : a = 20;   break;
7               case 2 : a = 30;   break;
8               default: a = 100;  break L;
9            }
10           a++;
11           System.out.print("(" +j+ "," +a+ ") " );
12       }
13       System.out.print("- Valor final de a = " + a);
14    }
15 }
```

Observe que o **break** com especificação de rótulo na linha 8 causa o término do **for** envolvente da linha 4, enquanto que os demais **break**s finalizam o comando **switch**.

O resultado impresso é: `(0,21) - Valor final de a = 100`.

1.9.11 Chamada de método

A chamada de um método tem o formato **x.f(***args***)**, onde **x** designa o objeto receptor da chamada, **f** é o nome do método a ser executado e **args** sua lista de argumentos.

1.9.12 Comando `return`

O comando **return** tem o formato **return** *expressão* para retorno de funções. Se a função retorna **void**, *expressão* deve ser omitida.

1.9.13 Comando `throw`

O comando **throw** tem o formato **throw** *expressão* e serve para levantar a exceção definida pela *expressão*.

1.9.14 Comando `try`

A habilitação e tratamento de exceção em Java é obtida pelo comando `try`, que tem o formato: `try bloco` *lista-de-catches*. O uso e tratamento de exceções estão detalhados no Capítulo 7.

1.9.15 Comando `synchronized`

O comando para sincronização de acesso de *threads* concorrentes a regiões críticas tem o formato:

`synchronized (`*expressão*`)` {*comando-associado*}

onde *expressão* é um argumento cujo valor deve ser uma referência para algum objeto.

Os comandos `synchronized` cujos argumentos apontam para um mesmo objeto atuam com sincronizador do acesso a regiões críticas, assegurando que dentre as várias *threads* concorrentes que os executaram somente uma delas de cada vez adquire o direito de prosseguir a execução do *comando-associado*. As demais *threads* aguardam sua vez em uma fila associada ao objeto referenciado pela *expressão* especificada no comando. Toda vez que uma *thread* deixa o corpo do comando `synchronized` há o desbloqueio das regiões críticas para uma das *threads* da fila assumir o controle e retomar a execução.

1.10 Entrada e saída básicas

A bibioteca padrão de Java para administar arquivos de fluxo de entrada e saída é muito rica, contendo cerca de 64 classes. Para entender perfeitamente o funcionamento dessas classes, um conhecimento mais avançado de Java do que apresenta aqui é necessário.

Contudo, reconhece-se que, para a implementação de programas que exerçam um mínimo de atração para um programador inici-

ante, é preciso tornar disponível o quanto antes recursos de fácil uso para realizar entrada e saída básicas de dados para tornar o processo de programação menos abstrato.

Assim, antecipa-se a apresentação da classe `JOptionPane` que faz parte da biblioteca `javax.wing`, a ser estudada em profundidade no Capítulo **??**, e que permite realizar interação simples com o usuário via uma interface gráfica elementar.

Descreve-se aqui também uma biblioteca básica não-padrão, denominada `myio`, projetada especialmente para ilustrar o uso da biblioteca de manipulação de fluxos de dados de Java e também para prover um pacote que facilite a programação de entrada e saída elementares.

A biblioteca `myio` implementa algumas operações fundamentais de arquivos de fluxo, envolvendo o teclado, a tela do monitor de vídeo e arquivos sem formatação armazenados em memória secundária.

As classes que compõem essa biblioteca são `InText`, `Screen`, `Kbd` e `OutText`. A implementação de cada uma dessas classes está detalhada na Seção 9.4.

1.10.1 Classe `JOptionPane`

A classe `JOptionPane` da biblioteca `javax.swing` oferece uma forma bastante simples de o programa interagir com o operador para troca de informações de pequeno volume. Essa classe possui um grande número de operações, dentre as quais destacam-se:

- `static int showConfirmDialog(null, Object m)`:
  exibe uma janela que pede confirmação `yes/no/cancel`. O valor retornado pode ser uma das constantes pré-definidas de `JOptionPane`: `YES_OPTION`, `NO_OPTION` ou `CANCEL_OPTION`.

- `static String showInputDialog(Object m)`:
  exibe uma janela com a mensagem `m` e permite a entrada de uma sequência de caracteres em um campo de texto editável. O valor digitado é retornado pela operação.
- `static void showMessageDialog(null,Object m,String t,int s)`:
  exibe uma janela com título `t` e mensagem `m` e aguarda um `OK` do operador para retornar. A janela possui um símbolo `s` definido por uma dentre as constantes:
  - `JOptionPane.ERROR_MESSAGE`
  - `JOptionPane.INFORMATION_MESSAGE`
  - `JOptionPane.WARNING_MESSAGE`
  - `JOptionPane.QUESTION_MESSAGE`
  - `JOptionPane.PLAIN_MESSAGE`

O primeiro parâmetro de muitos dos métodos de `JOptionPane` é do tipo `Component` e serve para indicar o componente gráfico da tela do monitor onde a janela de comunicação deve ser exibida. O uso de recursos gráficos é o tema tratado na Seção **??**. No momento, para simplificar, passa-se a `JOptionPane` um argumento com valor `null`, que faz a janela aparecer no centro da tela.

Figura 1.10 Mensagem informativa de JOptionPane

.

A execução do programa `Diálogo1` a seguir usa um dos métodos estático da classe `JOptionPane` para produzir a mensagem informativa da Fig. 1.10.

```
1 import javax.swing.JOptionPane;
2 public class Diálogo1 {
3    public static void main( String args[] ) {
4       JOptionPane.showMessageDialog(
5          null, "Welcome\nto\nJava\nProgramming!");
6       System.exit( 0 );
7    }
8 }
```

O programa **Adição**, abaixo, usa a operação **showInputDialog** para solicitar dois números inteiros e exibe de volta, por meio do método **showMessageDialog**, o valor da soma desses dois inteiros. A execução desse programa produz, em sequência, exibindo uma de cada vez, as janelas mostradas na Fig. 1.11.

```
1 import javax.swing.JOptionPane;
2 public class Adição {
3   public static void main( String[] args) {
4     String s1, s2; int sum;
5     s1 = JOptionPane.showInputDialog
6                          ("Enter first integer");
7     s2 = JOptionPane.showInputDialog
8                          ("Enter second integer");
9     JOptionPane.showMessageDialog(
10       null, "The sum is " + sum, "Results",
11                         JOptionPane.PLAIN_MESSAGE);
12    System.exit( 0 );
13  }
14 }
```

1.10.2 Classe `myio.Kbd`

Os métodos de **Kbd** da biblioteca **myio** têm nome segundo o padrão **read***tipo*, onde *tipo* pode ser **Int**, **Long**, **Float**, **Double** ou **String**.

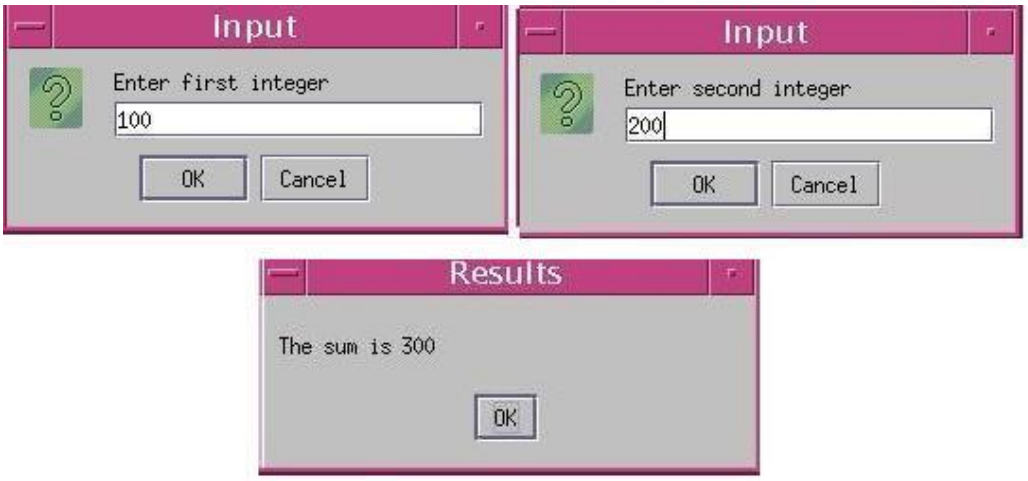


Figura 1.11 Interação com JOPtionPane

Esses métodos permitem abrir o teclado para leitura e converter os caracteres digitados para o tipo indicado no seu nome. São ignorados os espaços em branco e caracteres ascii de controle que por ventura forem digitados antes da sequência de caracteres a ser lida. A sequência termina quando for digitado um caractere branco ou de controle. Se durante o processo de leitura, algum tipo de erro for detectado, levanta-se a exceção `IOException`.

A sequência lida é automaticamente convertida para `int`, `float`, `double` ou `String`, conforme cada um dos métodos de leitura:

- `final static public boolean readBoolean() throws IOException`
- `final static public int readInt() throws IOException`
- `final static public long readLong() throws IOException`
- `final static public float readFloat() throws IOException`
- `final static public double readDouble()throws IOException`
- `final static public String readString() throws IOException`

O trecho de código abaixo mostra a leitura de um valor `int`, seguida da leitura de um `float`.

```
int   x = Kbd.readInt();
float f = Kbd.readFloat();
```

1.10.3 Classe `myio.InText`

Os métodos da classe **`InText`** funcionam como os de **`Kbd`**, exceto que os caracteres são lidos do arquivo cujo nome foi informado ao método construtor da classe, que é o responsável por tomar as providências para a abertura do arquivo. Esses métodos são:

- `public InText(String fileName) throws IOException`: função construtora responsável pela abertura do arquivo de nome `fileName`.
- `final public boolean readBoolean()throws IOException`
- `final public int readInt() throws IOException`
- `final public byte readByte( ) throws IOException`
- `final public long readLong() throws IOException`
- `final public float readFloat() throws IOException`
- `final public double readDouble() throws IOException`
- `final public String readString( ) throws IOException`
- `final public void close( ) throws IOException`

O trecho de programa abaixo mostra a leitura de um **`int`** e de um **`double`** do arquivo `C:\MeusArquivos\dados.txt`:

```
1 InText in = new InText("C:\MeusArquivos\dados.txt");
2 int    i = in.readInt();
3 double d = in.readDouble();
```

1.10.4 Classe `myio.Screen`

Os métodos da classe **`Screen`** têm um dos formatos

```
final static public void print(T x)
final static public void println(T x)
```

onde `T` pode ser **`String`**, **`boolean`**, **`byte`**, **`char`**, **`int`**, **`long`**, **`float`** ou **`double`**, e convertem o parâmetro **`x`** para uma sequência de caracteres ascii e a envia para o monitor de vídeo, para exibição a partir da posição corrente do cursor. Os métodos cujos nomes

terminam por **ln** (ele n) forçam uma mudança de linha após a escrita da sequência.

O programa **TestKS** é um exemplo completo, que mostra o uso das classes **Kbd** e **Screen**.

```
1  public class TestKS {
2    public static
3    void main(String[] args) throws IOException {
4       boolean b = false;  int x = 0;  long r = 0;
5       float y =(float) 0.0;  double d = 0.0;
6       String s = "indefinido";
7       Screen.println("Entre com os valores:");
8       Screen.print("boolean b = "); b = Kbd.readBoolean();
9          Screen.print("b lido = "); Screen.println(b);
10      Screen.print("int x  = "); x = Kbd.readInt() + 1;
11         Screen.print("x + 1 = "); Screen.println(x);
12      Screen.print("long r  = "); r = Kbd.readLong() + 1;
13         Screen.print("r + 1 = "); Screen.println(r);
14      Screen.print("float y = ");
15         y = Kbd.readFloat() + (float) 1.0;
16         Screen.print("y + 1.0 = " ); Screen.println(y);
17      Screen.print("double d = ");
18         d = Kbd.readDouble() + 1.0;
19         Screen.print("d + 1.0 = "); Screen.println(d);
20      Screen.print("String s = "); s = Kbd.readString();
21         Screen.print("s = "); Screen.println(s);
22   }
23 }
```

1.10.5 Classe `OutText`

Os métodos de **OutText** são análogos aos de **Screen**, exceto que a saída é redirecionada para o arquivo de fluxo criado pela função construtora da classe. O trecho de programa abaixo escreve no arquivo `C:\MeusArquivos\saída.txt` a sequência `"\n10\n11"`, onde

`\n` denota o caractere de mudança de linha.

```
int i = 10, k = 11;
OutText o = new OutText("C:\MeusArquivos\saída.txt");
o.println(i);
o.print(k);
```

1.10.6 Fim de arquivo

Diversas situações de erro podem ocorrer durante as operações de entrada e saída, mas, para manter o uso do pacote `myio` o mais simples possível, todos os erros são unificados em uma única exceção `IOException`. Se essa exceção não for devidamente tratada, o programa será cancelado. O usuário que não quiser tratá-la de forma elaborada deve apenas anunciá-la no cabeçalho das funções que usam a biblioteca `myio`, como mostra o programa:

```
1  public class TestInText {
2    public static
3    void main(String[] args) throws IOException {
4      boolean b = false;byte e = 0; char c = 'A'; int x = 0;
5      long r = 0; float y =(float) 0.0;  double d = 0.0;
6      String s = "indefinido";
7      InText in = new InText(args[0]);
8      b = in.readBoolean();
9        Screen.println("boolean b  = " + b);
10     e = in.readByte();  Screen.println("byte e  = " + e);
11     x = in.readInt();   Screen.println("int x  = " + x);
12     r = in.readLong();  Screen.println("long r  = " + r);
13     y = in.readFloat(); Screen.println("float y  = " + y);
14     d = in.readDouble();Screen.println("double d  = " +d);
15     s = in.readString();Screen.println("String s  = " +s);
16     in.close();
17   }
18 }
```

O erro mais provável é a leitura indevida da marca de fim de arquivo. Para evitar o cancelamento do programa por esse tipo de erro, o programador pode acrescentar uma marca especial no fim dos seus dados e explicitamente testá-la após cada operação de leitura, ou então implementar o tratamento da exceção `IOException`, embora outros tipos de erro possam ser mascarados pela mensagem emitida. O programa a seguir exemplifica o tratamento desse tipo de erro.

```
1  public class TestInTextEof {
2    public static void main(String[] args) {
3      boolean b=false;
4      byte e=0;
5      char c='A';
6      int x=0;
7      long r = 0;
8      float y =(float) 0.0;
9      double d = 0.0;
10     String s = "indefinido";
11     try {
12      InText in = new InText(args[0]);
13      b=in.readBoolean();Screen.println("boolean b = "+b);
14      e=in.readByte();  Screen.println("byte e = " + e);
15      x=in.readInt();   Screen.println("int x  = " + x);
16      r=in.readlong();  Screen.println("long r  = " + r);
17      y=in.readFloat(); Screen.println("float y  = " + y);
18      d=in.readDouble();Screen.println("double d  = " +d);
19      s=in.readString();Screen.println("String s = " + s);
20      in.close();
21     }
22     catch(Exception ex) {
23        Screen.println("Fim de arquivo!?");
24     }
25   }
26 }
```

1.11 Ambientes e escopo de nomes

Identificadores servem para dar nomes a diferentes elementos de um programa. Diferentes elementos podem ter o mesmo nome, desde que o compilador seja capaz de diferenciá-los pelo contexto. Para isto, há em Java dois mecanismos.

O primeiro mecanismo para identificação de nomes é a regra de escopo das declarações: declarações de nomes não-públicos em um pacote têm visibilidade restrita ao pacote, atributos privados somente são visíveis na classe onde foram declarados, o escopo de uma variável de método é o bloco onde foi declarada e o escopo de parâmetros é o corpo de seu método.

O segundo mecanismo, chamado **Ambiente de nomes**, particiona o universo de nomes conforme seu uso. Dentro de cada ambiente, nomes não podem ser repetidos, mas em ambientes distintos, não há restrição. São os seguintes os ambientes ou espécies de nomes definidos em Java:

- nomes de pacotes
- nomes de tipos (interfaces e classes)
- nomes de funções ou métodos
- nomes de campos de classe
- nomes de parâmetros e de variáveis locais a métodos
- nomes de rótulos de comandos

O programador deve escolher os nomes para os elementos do programa cuidadosamente e evitar repetições desnecessárias. Ou ainda melhor, usar repetições somente quando isso melhorar a legibilidade do programa.

Para ilustrar essa recomendação, veja que, embora permitido, veja um contra-exemplo, no qual um mesmo identificador é usado para designar muitos elementos distintos, como mostrado no pro-

grama abaixo. Certamente, essa prática não é recomendável.

```
1  package X;
2  class X {
3     static X X = new X( );
4     X( ){ }
5     X X(X X) {
6         X: while (X == X.X(X)) {
7             if (X.X.X != this.X) break X;
8         }
9         return new X( );
10    }
11 }
```

1.12 Conclusão

Este capítulo apresentou uma visão geral dos elementos fundamentais de linguagem Java, que são os tipos primitivos, expressões, comandos e operações elementares de entrada e saída de dados. Nos próximos capítulos, mecanismos elaborados para estruturação de programas são descritos.

Exercícios

1. Em Java a ordem de avaliação dos operandos corresponde ou não à ordem de aplicação dos operadores em uma expressão? Se não corresponder qual seria essa ordem?
2. Há linguagens, como o Pascal, que são orientadas por comandos. Outras, como Algol 68, são orientadas por expressões. Estude esses conceitos e determine como Java poderia ser classificada.
3. Por que efeito colateral em expressões pode ser nocivo à legibilidade do programa?

4. Como fazer para evitar expressões com efeito colateral?
5. Explique o significado de semântica de valor e de semântica de referência.
6. O comando `switch` envolve a série comparações da váriavel do `switch` com as constantes de cada `case`. Se houver muitos casos, esse processo pode ser lento. Qual seria uma solução alternativa?
7. Qual é o valor de `i` impresso pelo programa:

```
1 public class Avaliacao {
2   public static void main(String[] args) {
3     int i = -1;
4     int[] v = new int[8];
5     v[i+=1] = ((i==0) ? v[i+=2] : v[i+=1] - v[i+=4]);
6     System.out.println("\nValor de i = " + i);
7   }
8 }
```

Notas bibliográficas

As mais importantes referências para a linguagem Java são as páginas da Oracle (http://www.oracle.com), que inclui tutoriais de fácil leitura, e os livros publicados por James Gosling, Ken Arnold et alii [1, 2, 3, 20, 22]. Para detalhes de Java SE 14, veja [22].

O livro dos Deitels [10] é uma boa referência. Esse livro apresenta um grande volume de exemplos esclarecedores do funcionamento da linguagem Java.

Capítulo 2

Classes e Objetos

Objetos são elementos de software que representam entidades de uma aplicação. O mundo real que se deseja modelar é composto de objetos, e o software que o modela deve implementar esses objetos de forma a que representem os da vida real.

Objetos manipulados por um programa são abstrações de dados, implementadas por meio de uma estrutura de dados e de um conjunto de operações. Assim, objetos têm uma representação interna, definida por uma estrutura de dados, e seus estados são caracterizados pelos valores das variáveis que compõem essa estrutura.

Objetos devem ser manipulados somente por suas operações. Manipulação direta da estrutura de dados que representa um objeto viola o princípio da abstração de dados. Mesmo que sua linguagem de programação preferida permita isto, as boas práticas da Engenharia de Software recomendam o encapsulamento da representação dos objetos e sua manipulação exclusivamente via as operações definidas para esse fim.

Objetos em um programa orientado por objetos correspondem às tradicionais variáveis. Da mesma forma que variáveis têm tipo, segundo o qual elas devem ser alocadas e operadas, objetos são criados a partir da especificação de sua classe, que provê informações relativas ao tipo do objeto.

Assim, classe é um mecanismo para implementar o tipo de obje-

tos, sendo dotada de recurso linguístico apropriado para descrever o comportamento de um conjunto de objetos. Classe é, portanto, um molde para construir objetos. Para isso, classes oferecem mecanismos para declarar a representação dos objetos e definir as operações a eles aplicáveis.

A implementação das operações e das estruturas de dados internas de um objeto devem ser protegidas contra acessos externos para que se possa garantir a integridade dos objetos e facilitar o entendimento de seu comportamento. Associados a classes há os conceitos de Abstração, Encapsulação, Proteção de Dados, Ligação Dinâmica e Herança .

Abstração trata dos mecanismos que permitem que se concentre nas partes relevantes do problema e que se ignore o que não for relevante no momento. Literalmente, *abstrair* significa *retirar*, e no jargão da programação modular, *abstrair* significa retirar o detalhe do centro das atenções, para facilitar a compreensão do todo. Programar compreende identificar as abstrações pertinentes a uma dada aplicação e usar classes para a sua concretização.

O mecanismo mais importante para implementar abstrações é **encapsulação**. Encapsular é, em primeiro lugar, segundo o Dicionário, o ato de colocar alguma coisa dentro de uma cápsula, sem necessariamente impedir-lhe o acesso. Em orientação por objetos, o termo comumente incorpora também a idéia de **proteção de dados** contra acessos indevidos, realizados à margem das operações definidas para a classe.

Ligação dinâmica de nomes a funções implementadas é um mecanismo inerente a orientação por objetos, o qual permite que somente em tempo de execução a função a ser executada para uma dada invocação seja determinada.

Herança é um recurso linguístico que permite criar uma nova

classe por meio da especialização de outra. Herança possibilita o reúso de código e a criação de hierarquia de tipos.

2.1 Criação de novos tipos

Uma classe define um novo tipo de dados. E os principais elementos constituintes de uma classe são: membros de dados, também chamados de atributos, funções construtoras, função finalizadora, blocos de iniciação e outras funções ou métodos.

Os membros de dados definem a representação dos objetos da classe. O relacionamento entre uma classe e seus membros de dados é o de composição, isto é, a classe é formada pela composição de membros de dados. Esse relacionamento de composição é denominado **tem-um**.

Funções construtoras servem para automaticamente iniciar o estado dos objetos no momento de sua criação. Funções construtoras são como os demais métodos, exceto que não podem ser chamadas diretamente. Elas estão vinculadas à criação de objetos.

A função finalizadora é uma função especial que é automaticamente invocada no momento em que a área de um objeto estiver sendo recolhida pelo coletor de lixo.

As operações da classe são definidas por seus membros-função. Essas operações devem prover a forma exclusiva de acesso e manipulação da representação dos objetos da classe.

A classe `PontoA` apresentada a seguir define um tipo abstrato de dados `PontoA`, caracterizado pelas operações `set`, `getX`, `getY`, `clear` e `display`. As funções nomeadas `PontoA` são funções construtoras.

O caráter abstrato do tipo definido deve-se ao fato de os campos `x` e `y` terem sido declarados de acesso privativo.

```
1  public class PontoA {
2    private int x, y;
3    public PontoA(int a,int b) {this.x=a; this.y=b;}
4    public PontoA() {this(0,0);}
5    public set(int a,int b) {this.x=a; this.y=b;}
6    public int getX() {return this.x;}
7    public int getY() {return this.y;}
8    public void clear() {this.x = 0; this.y = 0;}
9    public String display() {
10     return "(" + this.x + "," + this.y + ")";
11   }
12 }
```

A classe `PontoA` pode então ser usada como molde para construir objetos, os quais compartilham o mesmo leiaute para os seus campos, conforme ilustra a Fig. 2.1.

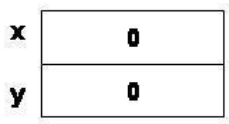


Figura 2.1 Leiaute de `PontoA`

As operações definidas na classe `PontoA` atuam sobre os campos dos respectivos objetos, que são alocados e iniciados no momento da sua criação pelo operador `new`. No corpo dessas operações, o endereço do objeto sobre o qual se aplicam em cada invocação é identificado pela referência implicitamente definida `this`.

```
1  public class PontoB {
2     private int x, y;
3     public PontoB(int a, int b) {x = a; y = b;}
4     public void clear() {this.x = 0; this.y = 0;}
5  }
```

Por economia de escrita, a referência `this`, em muitas situações, pode ser omitida, como mostra a declaração de `PontoB` acima.

As ocorrências de `x` e `y` na linha 3 da classe acima devem ser entendidas como `this.x` e `this.y`, respectivamente.

2.2 Criação de objetos

A área de dados de um programa normalmente consiste em duas partes: pilha de execução e *heap*. As variáveis locais a métodos são automaticamente alocadas na pilha de execução, no início da execução do método que as contém, e liberadas no ato de retorno do método. O mesmo vale para os parâmetros do método. Na pilha de execução somente são alocados áreas para variáveis de tipos básicos e referências. Objetos devem ser criados explicitamente com base na descrição provida por suas classes e alocados apenas no *heap*.

Note que se `A` for uma classe, uma declaração da forma `A a` apenas cria uma referência `a` capaz de vir a apontar para um objeto do tipo `A`. Uma declaração da forma `A[] b` apenas cria uma referência `b` para objetos do tipo arranjo de elementos cujo tipo é referência para objetos do tipo `A`. Graficamente, tem-se a estrutura da Fig. 2.2 para essas declarações:

Figura 2.2 Referências

A criação de objetos a partir de uma classe `A` dá-se pela avaliação da expressão `new A(`*args*`)`, que realiza as seguintes ações:

- alocação, na área do *heap*, do objeto representado pela estrutura de dados descrita pelos membros de dados da classe `A`
- iniciação dos campos do objeto com valores *default*, conforme seus tipos

- iniciação dos campos conforme expressão de iniciação definida para cada um na sua declaração, se for o caso
- execução da função construtora com os argumentos *args*
- retorno do endereço da área alocada, que tem tipo referência a objetos do tipo `A`.

Por exemplo, a declaração `PontoA a = new PontoA(10,20)` causa a criação de um objeto do tipo referência para `PontoA`, a alocação da referência `a`, iniciada com o endereço do objeto `PontoA` criado, como mostra Fig. 2.3.

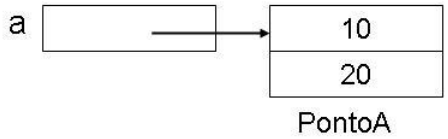


Figura 2.3 Alocação de objeto `PontoA`

A criação de objetos do tipo arranjo é obtida pela avaliação de `new A[` t `]`, onde t é uma expressão que dá o tamanho do arranjo a ser alocado e `A`, o tipo de seus elementos. Essa operação aloca o arranjo do tamanho indicado, inicia seus campos com valores *default* e retorna o endereço da área alocada.

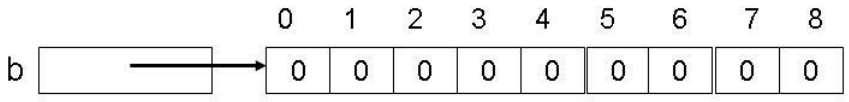


Figura 2.4 Alocação de arranjo de Int

Por exemplo, a declaração `int[] b = new int[9]` define `b` como uma referência para vetores de inteiros, aloca um vetor de 9 inteiros, zera todos os seus elementos e armazena o endereço da área alocada na referência `b`, produzindo a alocação da Fig. 2.4.

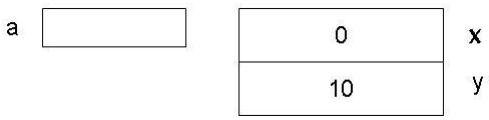
Para ver o processo de criação de objetos em detalhes, considere o programa ilustrativo `Criação`, que cria três objetos da classe `M`, que são referenciados por `a`, `b` e `c`.

```
1  public class Criação {
2     public static void main(String[] args) {
3        M a = new M();
4        M b = new M();
5        M c;
6        c = b;
7        c.x = b.x + 1;
8        a = b;
9        a.y = b.y + c.y;
10    }
11 }
12 class M {
13    public int x;
14    public int y = 10;
15 }
```

A execução desse programa inicia-se na linha 3 da classe `Criação` com o processamento da declaração `M a`, que causa a alocação, na pilha de execução, da referência `a`, com valor indefinido, produzindo:

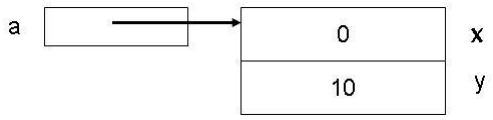
Diferentemente de campos de objetos, que são sempre alocados no *heap*, variáveis locais a métodos, que são alocadas na pilha de execução, nunca são iniciadas automaticamente. Contudo, o compilador verifica o código e exige que haja uma iniciação evidente de toda variável local no texto do método.

A seguir, ainda na linha 3, a operação `new M()` cria um objeto do tipo `M` na área do *heap*, iniciando o campo `x` com o valor default `0`, e o campo `y`, com o valor indicado em sua declaração:

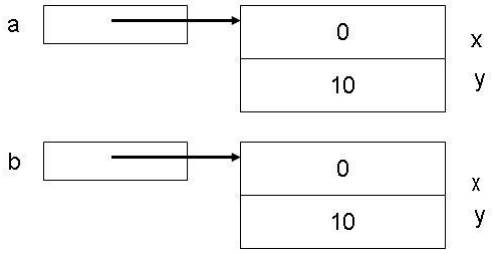


A execução do comando da linha 3 conclui-se com a atribuição

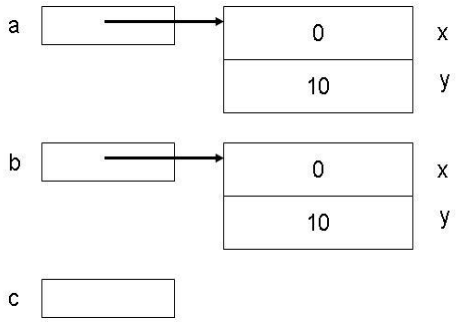
do endereço do objeto alocado e inicializado à referência `a`, produzindo o seguinte mapeamento da memória:



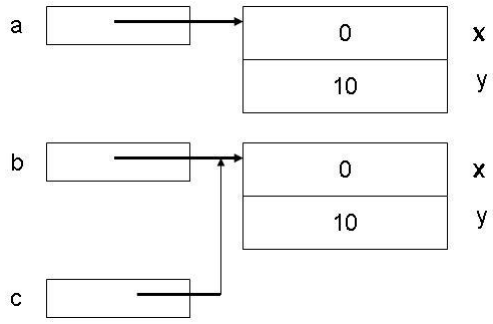
A execução da linha 4 causa a alocação de um novo objeto, também do tipo `M`, e armazena seu endereço na referência declarada `b`, conforme mostra-se:



A declaração da linha 5 cria uma referência `c` de valor inicial indefinido:

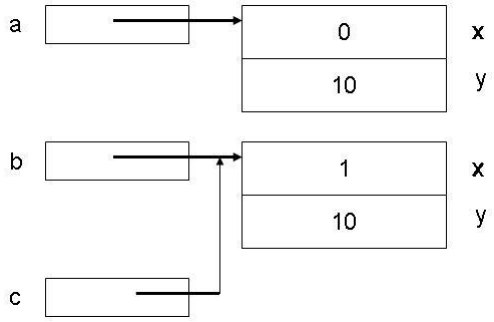


O comando de atribuição da linha 6 cria dois caminhos de acesso a um mesmo objeto, via as referências `b` e `c`, criando um compartilhamento.

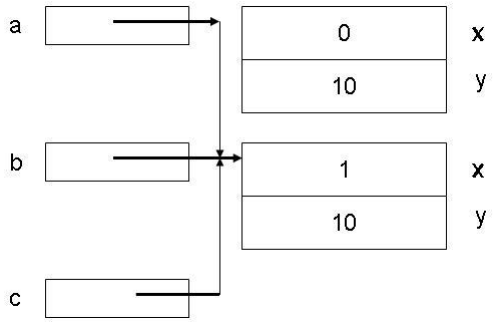


Para se fazer cópias de objetos, deve-se usar o método `clone` definido na classe `Object`, conforme detalhado no Capítulo 16.

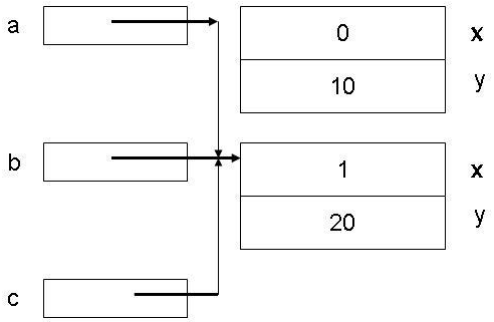
A linha 7 mostra que há dois caminhos de acesso aos campos do objeto apontado por `b` e `c`. O resultado, após executar o comando de atribuição dessa linha, é:



Na linha 8 um novo compartilhamento é criado. Agora, o mesmo objeto é referenciado por `a`, `b` e `c`:



O resultado final, após executar o comando da linha 9, é:



2.3 Controle de visibilidade

Há pelo menos dois níveis importantes de controle de visibilidade disponíveis na linguagem Java. No primeiro nível, estão as declarações de variáveis locais a métodos e de seus parâmetros. Es-

ses elementos têm visibilidade restrita ao corpo do método que os contém. Em segundo nível, tem-se a visibilidade dos membros de classes, que pode estar restrita à respectiva classe, a suas subclasses ou ao pacote que contém a sua classe.

A definição de pacote pode ser encontrada no Capítulo 11. No momento é suficiente saber que um pacote é uma coleção de classes que detêm algum privilégio de acesso entre si, e que o propósito de um pacote é estabelecer uma fronteira de visibilidade.

Assim, métodos, classes e pacotes são mecanismos de Java que permitem encapsular seus elementos constituintes, exercendo sobre eles controle de sua visibilidade em outras partes do programa, que é um recurso indispensável para se atingir modularidade.

Os principais constituintes, i.e., atributos, métodos e construtoras, de uma classe têm sua visibilidade definida prefixando suas declarações por um dos seguintes modificadores:

- `public`: membros declarados como `public` são acessíveis de qualquer lugar de onde a classe for acessível, inclusive nas respectivas subclasses.
- `private`: membros declarados como `private` somente são acessíveis aos membros da própria classe.
- `protected`: membros declarados como `protected` são membros accessíveis somente na própria classe, em suas subclasses e por outras classes que estiverem dentro do mesmo pacote. Diz-se que esses membros têm visibilidade pacote.
- *vazio*: membros declarados sem modificador de acesso são acessíveis somente por classes que estiverem no mesmo pacote.

Para ilustrar o uso de modificadores de acesso, considere as classes `Ponto`, que tem todos os seus membros declarados públicos, `TrincaDePontos`, com campos de visibilidade privada, protegida e pacote, e uma terceira classe, de nome `Visibilidade1`, também

declarada no mesmo pacote, que tem acesso aos elementos dessas classes, excetuando-se o campo `p1`, por ele ter sido declarado `private`. O erro indicado na linha 24 deve-se a esse fato.

```
1  class Ponto {
2     public double x, y;
3     public void clear() {x = 0; y = 0;}
4  }
5  class TrincaDePontos {
6     private Ponto p1;  // visível localmente
7     protected Ponto p2;// visível nos descendentes e pacote
8     Ponto p3;          // visível no pacote
9     public TrincaDePontos() {
10       p1 = new Ponto();
11       p2 = new Ponto();
12       p3 = new Ponto();
13    }
14    public void clear() {
15       p1.clear(); p2.clear(); p3.clear();
16    }
17 }
18 class Visibilidade1 {
19    public static void main(String[] args) {
20       Ponto a = new Ponto();
21       TrincaDePontos p = new TrincaDePontos();
22       Ponto b, c, d;
23       a.x = a.y + 1; // OK
24       b = p.p1;      // ERRADO
25       c = p.p2;      // OK
26       d = p.p3;      // OK
27       p.clear();     // OK
28    }
29 }
```

Se a classe `Visibilidade2`, apresentada a seguir, residisse em um pacote distinto do das classes `Ponto` e `TrincaDePontos`, erros seriam também detectados nas suas linhas 7, 8 e 9.

```
1  public class Visibilidade2 {
2     public static void main(String[] args) {
3        Ponto a = new Ponto();
4        TrincaDePontos p = new TrincaDePontos();
5        Ponto b, c;
6        a.x = a.y + 1; // OK
7        b = p.p1;      // ERRADO
8        c = p.p2;      // ERRADO
9        d = p.p3;      // ERRADO
10       p.clear();     // OK
11    }
12 }
```

2.4 Métodos

Em Java, métodos somente podem ser declarados no corpo de classes e correspondem ao conceito das tradicionais funções em linguagens de programação, por isso, são também chamadas de funções membros. Métodos podem ou não retornar valores. Métodos que não retornam valores são chamados de funções `void`, pois esse é o tipo a ser declarado para o seu valor de retorno.

Os parâmetros de métodos são alocados na pilha de execução no momento que o método é chamado e liberados no seu retorno. Os parâmetros são iniciados com os valores dados pelos respectivos argumentos de chamada. Isso chama-se passagem **por valor**, que é o único modo de passagem de parâmetro disponível em Java. Nesse modo, o valor do parâmetro de chamada é copiado para a área do parâmetro formal, que se comporta como uma variável local ao método. Note que ao passar a referência de um objeto como parâmetro, somente o valor da referência é efetivamente copiado para a área de execução do método. O objeto referenciado nunca é copiado, seja ele um arranjo ou um objeto qualquer.

Não há passagem de parâmetros por referência em Java, mas referências podem ser passadas por valor. Estando dentro de um método, é possível, via a referência dada por um parâmetro, modificar o objeto externo por ela apontado. Entretanto, o valor do parâmetro de chamada passado, não pode ser modificado, conforme ilustram os programas `Passagem1` e `Passagem2` abaixo.

```
1  public class Passagem1 {
2     public static void main(String [] args) {
3        int x = 1 ;
4        System.out.println("A: x = " + x);
5        incrementa(x);
6        System.out.println("D: x = " + x);
7     }
8     public static void incrementa(double y) {
9        y++;
10       System.out.println("M: y =" + y);
11    }
12 }
```

O programa `Passagem1` imprime: `A: x = 1 M: y = 2 D: x = 1`, e o programa `Passagem2` produz a saída: `(10,10) (0,0) 1 2 4`.

```
1  public class Passagem2 {
2     public static void altera1(Ponto p) {p.clear();}
3     public static void altera2(int[] t) {t[2] = 4;}
4     public static void main(String [] args) {
5        PontoA q = new PontoA();
6        int [] y = {1, 2, 3};
7        q.set(10.0,10.0);  q.print();
8        altera1(q);  q.print(); altera2(y);
9        for (int z: y) System.out.print(" " + z);
10    }
11 }
```

A operação de chamar uma das funções de uma classe qualificando-a por um dado objeto, por exemplo, `p.clear()`, é interpretada

como enviar a mensagem, no caso `clear`, ao objeto receptor, nesse exemplo, `p`. O objeto receptor é aquele que recebe e trata a mensagem. Tratar uma mensagem é executar a função membro que corresponde à mensagem recebida. Essa visão é importante, pois pode haver mais de uma função correspondendo a uma mesma mensagem, cabendo ao sistema de execução selecionar aquela que for aplicável em cada caso. Esse processo, que é denominado **ligação dinâmica**, implementa um importante tipo de polimorfismo.

2.5 Funções construtoras

Objetos são alocados no *heap* pelo operador `new`, e seus campos são automaticamente *zerados*, conforme seus respectivos tipos. Campos numéricos recebem valor `0`, booleanos, `false`, e as referências a objeto são iniciados com o valor `null`. Ainda durante a execução do operador `new`, as iniciações indicadas nas declarações dos atributos da classe são processadas, e, por último, o corpo de uma das funções construtoras declaradas na classe, selecionada conforme os tipos dos parâmetros, é executada.

Uma função construtora distingue-se dos demais métodos de uma mesma classe por ter o nome da classe e não possuir especificação de tipo de retorno. A presença de funções construtoras em uma classe não é obrigatória. Por outro lado, pode-se especificar mais de uma função construtora em uma mesma classe, desde que elas sejam distinguíveis uma das outras pelo número de parâmetros ou pelos tipos de seus parâmetros.

Na prática, os corpos das funções construtoras de uma mesma classe guardam obrigatoriamente uma relação entre si, no sentido de que todas têm a função comum de iniciar o estado de cada objeto no momento de sua alocação.

Com frequência, as funções construtoras com menos argumentos, dentro de um conjunto de construtoras, são casos particulares de iniciação em que os argumentos faltantes têm valores *default*. Assim, para evitar repetição de código, as boas práticas de programação recomendam que se escreva o corpo detalhado apenas da construtora com mais parâmetros, e a implementação das demais resume-se em chamar uma das construtoras já implementadas com parâmetros apropriados.

Construtoras não podem ser chamadas diretamente, deve-se fazê-lo via chamada à função especial `this`. Na verdade, a palavra-chave `this`, em Java, serve a dois propósitos: (i) denotar o endereço do objeto receptor de uma mensagem; (ii) representar a função construtora identificada a partir de seus parâmetros.

A chamada de construtora via `this` deve ser o primeiro comando no corpo da respectiva construtora, como é feito no programa:

```
1 public class Construtora1 {
2    public Construtora1() {this(default1, default2);}
3    public Construtora1(Tipo1 arg1) {this(arg1,default2);}
4    public Construtora1(Tipo1 arg1, Tipo2 arg2) {...}
5 }
```

O programa abaixo mostra o uso de cada uma das construtoras da classe `Construtora1`. A construtora ativada em cada caso é aquela determinada pelo número e tipo de parâmetros.

```
1 public class TesteDeConstrutora {
2    Tipo1 a; Tipo2 b;
3    Construtora1 x = new Construtora1();
4    Construtora1 y = new Construtora1(a);
5    Construtora1 z = new Construtora1(a,b);
6 }
```

O programa `Janela` abaixo mostra com mais detalhes o processo de alocação e iniciação de objetos.

```
1  class Ponto {
2     private double x, y;
3     public Ponto() {this(0.0, 0.0);}
4     public Ponto(double a, double b) {x=a; y=b;}
5     public void set(double a, double b) {x=a; y=b;}
6     public void hide() { ... }
7     public void show() { ... }
8  }
9  public class Janela {
10      Ponto esquerdo  = new Ponto();
11      Ponto direito = new Ponto(8,9);
12      Ponto meio = new Ponto(4,5);
13      Ponto z = new Ponto();
14      ...
15 }
```

A execução da declaração `Janela j = new Janela()` em alguma outra classe dentro do escopo das declarações acima produz a alocação na Fig. 2.5.

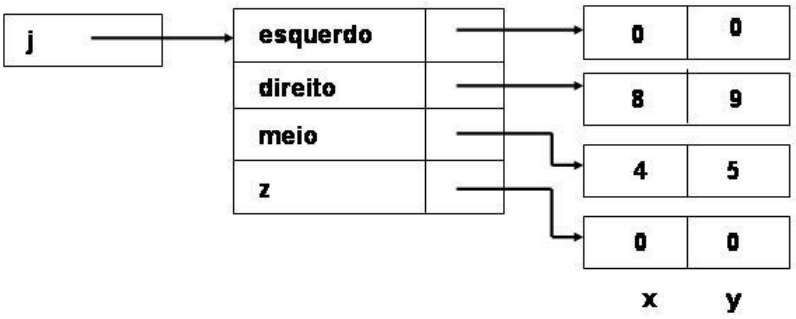


Figura 2.5 Alocação da janela `j`

Funções construtoras estão sujeitas ao mesmo mecanismo de controle de visibilidade que se pode impor a qualquer elemento de uma classe. Normalmente funções construtoras são `public`. Uma função construtora com visibilidade `private` é possível, mas isso impede que objetos da classe sejam criados fora de sua classe. E se a função construtora for `protected`, somente a própria classe

ou seus herdeiros poderão criar objetos da respectiva classe.

Se não houver na classe definição alguma de construtoras, por *default*, é criada uma sem parâmetros.

2.6 Funções finalizadoras

Em Java, um objeto é dito não ter utilidade quando não houver mais qualquer referência válida para ele, e, portanto, tornar-se inacessível pelo programa. Sem utilidade, ele pode ser desalocado, de forma que a área que ocupa na memória *heap* possa ser liberada para ser reusada.

O sistema de coleta de lixo da JVM automaticamente administra a reciclagem das áreas de objetos sem uso, liberando o programador de ter que se preocupar com a recuperação dos espaços dos objetos inúteis. O coletor de lixo é um processo que é automaticamente executado pela JVM em segundo plano, e somente entra em ação quando o tamanho da área de espaço disponível atingir valor inferior ao considerado crítico.

Isto resolve parte do problema de gerência de recursos, mas há objetos que detêm controle sobre outros recursos do ambiente de execução, por exemplo, arquivos abertos, que devem ser devidamente fechados quando os objetos que os controlam encerrarem sua participação no programa. Entretanto, a menos que se faça um esforço específico de programação para administrar o tempo de vida de um determinado objeto, o usual é que não se saiba em um ponto qualquer durante a execução se esse objeto ainda está sendo referenciado ou se já teve sua participação no programa encerrada. Assim, a decisão de executar explicitamente uma operação sobre um objeto para que ele libere os recursos mantidos sob seu controle pode não ser fácil de ser tomada.

Para contornar essa dificuldade, Java possibilita a definição de uma função membro especial, de nome `finalize`, que é executada automaticamente quando a área de um objeto da classe que a contém estiver para ser liberada pelo coletor de lixo. Por exemplo, na classe `ProcessaArquivo` abaixo, a função `finalize`, quando chamada pelo coletor de lixo, providenciará o fechamento do arquivo que foi aberto na respectiva função construtora do objeto:

```
1  public class ProcessaArquivo {
2     private Stream arquivo;
3     public ProcessaArquivo(String nomeDoarquivo) {
4        arquivo = new Stream(nomeDoArquivo);
5     }
6     public void fechaArquivo() {
7        if (arquivo != null) {
8            arquivo.close(); arquivo.null();
9        }
10    }
11    protected void finalize() throws Throwable {
12       super.finalize(); fechaArquivo();
13    }
14 }
```

O método `finalize` é um método `protected` definido na classe `Object`. Toda classe Java é uma extensão direta ou indireta de `Object`, cujos métodos podem ser redefinidos em subclasses, obedecendo certas regras. Em particular, de acordo com sua assinatura em `Object`, `finalize`, a menos que sua visibilidade seja ampliada em uma subclasse, não lhe é permitido ser chamada diretamente pelo cliente da classe que o contém. Outros detalhes sobre esse tópico, por exemplo, o significado de `super`, que aparece na linha 12, podem ser encontrados no Capítulo 4.

Há um real perigo de a operação `finalize` ressuscitar seu objeto corrente, i.e., criar uma referência para ele, por exemplo, via

uma referência estática, e assim impedindo que o objeto seja recolhido pelo coletor de lixo. Todavia, embora permitida, ressurreição de objetos não é uma boa prática de programação, haja vista que Java invoca a função `finalize` no máximo uma vez para cada objeto. Consequentemente objetos ressuscitados, quando vierem a ser liberados novamente, são recolhidos pelo coletor de lixo sem chamada à função `finalize`. Isto é, objetos ressuscitados perdem direito à finalização, provavelmente complicando o entendimento do programa. As boas práticas recomendam que se faça um *clone* do objeto em vez de *ressuscitá-lo*.

Ressalta-se que não se tem garantia de que o coletor de lixo será ativado durante a execução do programa, pois a coleta de objetos sem uso pode nunca ocorrer e não se pode garantir que a função `finalize` do objeto seja sempre executada. Observe os valores impressos pelos programas `G1` e `G2` abaixo, nos quais, dois objetos são alocados, sendo um deles explicitamente liberado.

```
1  public class G1 {
2    public static void main(String[] args) {
3      A a = new A(); a = new A ();
4      System.gc();
5      for (int i=1; i <1000000; i++);
6      System.out.print("Acabou!");
7    }
8  }
9  class A {
10   protected void finalize() throws Throwable {
11      super.finalize();
12      System.out.print("Finalize foi chamada. ");
13   }
14 }
```

As funções `finalize` definidas nesses programas servem para revelar se foram elas chamadas pelo coletor de lixo, isto é, para

mostrar que os objetos abandonados pelos programas foram ou não recolhidos pelo coletor de lixo.

```
1  public class G2 {
2    public static void main(String[] args) {
3       A a = new A(); a = new A ();
4       //System.gc();
5       for (int i=1; i <1000000; i++);
6       System.out.print("Acabou!");
7    }
8  }
9  class A {
10   protected void finalize() throws Throwable {
11      super.finalize();
12      System.out.print("Finalize foi chamada. ");
13   }
14 }
```

Para se assegurar que o coletor de lixo seja de fato chamado, o programa `G1` o ativa explicitamente via comando `System.gc()`, na linha 4, demonstrado pela impressão de

`Saída: Finalize foi chamada. Acabou!`

Isto se faz necessário porque pequenos exemplos, como os mostrados, não demandam memória o suficiente para necessitar do concurso do coletor de lixo. O comando `for` da linha 5, iterando um milhão de vezes, foi usado para simular algum processamento mais pesado naquele ponto do programa.

No programa `G2`, o coletor de lixo não foi ativado, porque não se lhe fez uma chamada explícita nem seus serviços se fizeram necessários, dado a pequena demanda de memória do exemplo. O valor impresso por `G2` é `Acabou!`, mostrando que a função `finalize` não foi executada.

Para enfatizar esse argumento, o programa `G3` abaixo produz

`Saída: Finalize foi chamada`

mostrando que a chamada do coletor de lixo garante a execução da função `finalize` dos objetos liberados até o ponto dessa chamada.

```
1  public class G3 {
2    public static void main(String[] args) {
3       A a = new A(); a = new A ();
4       System.gc ();
5       //for (int i=1; i <1000000; i++);
6       //System.out.print("Acabou!");
7    }
8  }
9  class A {
10   protected void finalize() throws Throwable {
11      super.finalize();
12      System.out.print("Finalize foi chamada. ");
13   }
14 }
```

Conclui-se que a função `finalize` é um recurso de programação importante, mas um pouco complexo. A dificuldade de se garantir que sua execução irá ocorrer para todo objeto liberado torna seu uso perigoso e obscurece a semântica do programa, devendo seu uso ser evitado sempre que possível. Na maioria das aplicações é mais seguro implementar um método de finalização público e ordinário, controlar explicitamente a vida de determinados objetos e acionar o processo de finalização, quando for conveniente.

2.7 Referência ao receptor

Um dos usos da palavra-chave `this` é prover automaticamente o endereço ou referência do objeto corrente durante a execução de um método, i.e., o endereço do objeto receptor da mensagem associada ao método. Por exemplo, a execução do seguinte trecho de código:

```
Janela j = new Janela(); j.set();
```

da classe `Janela` definida a seguir, indica que, durante a execução do método `set`, definido na linha 5 da classe `Janela`, a variável `this`, válida no escopo de `set`, tem o mesmo valor de `j`, o qual é o endereço do objeto corrente. Assim, `this.esquerdo.x = 15` da linha 6 é o mesmo que `j.esquerdo.x = 15`.

```
1  class Janela {
2     private Ponto esquerdo = new Ponto();
3     private Ponto direito  = new Ponto();
4     private Ponto z;
5     public void set() {
6        this.esquerdo.x = 15; this.z = this.direito;
7        this.direito.x = 10;  this.direito.y = 20;
8        this.z.clear();
9     }
10 }
11 class Ponto {
12    public double x, y;
13    public void clear() {this.x = 0; this.y = 0;}
14 }
```

Similarmente, a chamada de `clear`, na linha 8 do método `set` do programa abaixo, tem como objeto receptor o `Ponto` apontado pelo campo `z` de objeto referenciado por `j`, conforme Fig. 2.6.

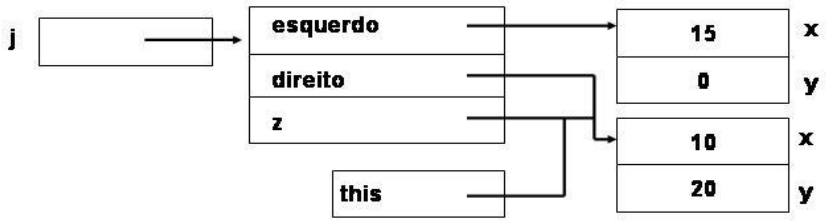


Figura 2.6 Configuração antes do `clear`

Para entender com mais clareza o funcionamento da referência `this`, considere a tradução de um método não-estático de Java para um código equivalente na linguagem de programação C. Nessa tradução, a referência `this` é tratada como o nome de um parâmetro

implícito adicional que tem como tipo a classe que contém o método e que recebe a cada chamada ao método o endereço do objeto corrente. Por exemplo, o método `clear` da linha 13 da classe `Ponto` pode ser traduzido para a linguagem C para

```
void clear(Ponto* this) {...}
```

E a chamada `z.clear()` deve ser traduzida para `clear(z)` de maneira a passar o valor de `z` para o parâmetro `this`.

A referência `this` pode ser omitida sempre que puder ser deduzida pelo contexto, mas deve ser usada quando for necessário explicitar as referências desejadas, como mostra o exemplo abaixo.

```
1  public class Rectangle {
2     private int x, y, largura, altura;
3     public Rectangle(int x, int y, int w, int h) {
4        this.x = x; this.y = y;
5        this.largura = w;
6        altura = h;
7     }
8     public void identify() {
9        System.out.print("Ret (" + x + "," + y + ") - ");
10       System.out.println("Dimensões: " + largura
11                                  + " X " + altura);
12    }
13    public static void main(String args[]) {
14       Rectangle r1 = new Rectangle(5,5,100,200);
15       Rectangle r2 = new Rectangle(0,9,340,250);
16       r1.identify();  r2.identify();
17    }
18 }
```

Observe que o uso de `this` na linha 4 é obrigatório para se ter acesso aos campos `x` e `y` do objeto corrente, uma vez que parâmetros têm visibilidade prioritária sobre atributos da classe. O `this` da linha 5 poderia ter sido omitido, como o foi na linha 6, onde escreveu-se `altura` no lugar de `this.altura`.

A Fig. 2.7 mostra a alocação dos objetos quando o fluxo de execução do programa `Rectangle`, a partir de sua função `main`, atingir o início do método `identify`, definido na linha 8, após a chamada `r1.identify()` na linha 16. Note que a referência `this`, nesse momento, aponta para o mesmo objeto correntemente referenciado por `r1`.

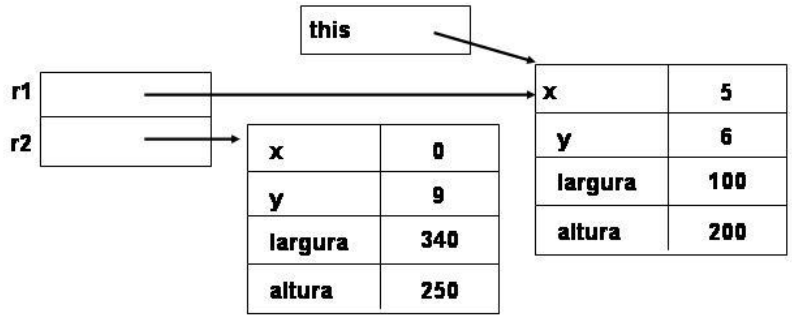


Figura 2.7 Configuração no início de `r1.identify()`

No fim da ativação de `r1.identify()`, o texto impresso é:

```
Ret (5,5) - Dimensões:  100 X 200
```

Analogamente para a chamada `r2.identify()`, tem-se a configuração da Fig. 2.8.

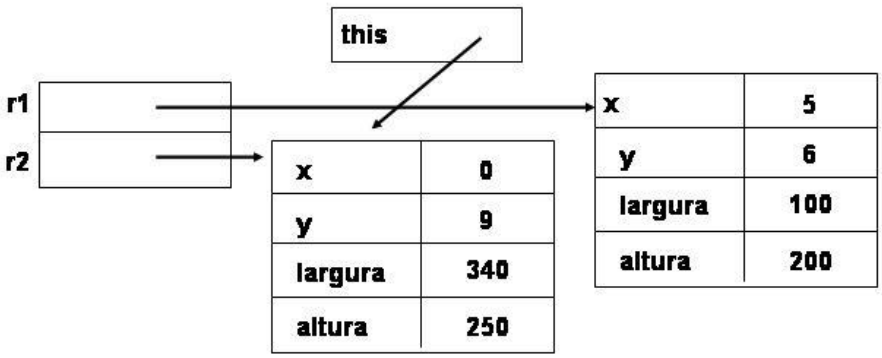


Figura 2.8 Configuração no início de `r2.identify()`

E o valor impresso é `Ret (0,9) - Dimensões 340 X 250`.

2.8 Variáveis de classe e de instância

Os atributos declarados em uma classe podem dar origem a variáveis de classe ou a variáveis de instância. Variáveis de instância são campos de cada objeto alocado, enquanto uma variável de classe é um campo do meta-objeto descritor da classe, e que é compartilhado por todos os objetos dessa classe. Para ilustrar as diferenças entre esses dois tipos de variáveis, considere a classe `Conta1` definida abaixo.

```
1 class Conta1 {
2    private String nome;
3    private String número;
4    private double saldo;
5    public  double saldoMinimo = 100.00;
6    public Conta1(double saldoMinimo) {
7       this.saldoMinimo = saldoMinimo;
8    }
9 }
```

Os objetos do tipo `Conta1`, que venham a ser criados, terão o leiaute formado pelo objeto e pelo descritor da classe, conforme Fig. 2.9.

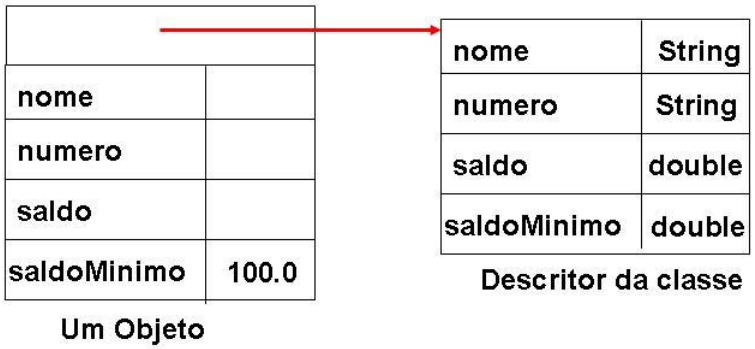


Figura 2.9 Leiaute de um objeto

O descritor da classe, que aparece nessa figura, é um objeto do tipo `Class`, que descreve a estrutura da classe `Conta1`. Há um

objeto do tipo `Class` para cada classe usada em um programa. Esse objeto é alocado no momento em que o nome de sua classe for encontrado durante a execução, por exemplo, na declaração de uma variável ou de um parâmetro.

Todo objeto automaticamente possui um apontador para o descritor de sua classe. Isto permite que se identifique o tipo de qualquer objeto durante a execução, via a operação

referência-ao-objeto `instanceof` *nome-de-uma-classe*

que retorna `true` ou `false`.

A aplicação `Banco1` abaixo utiliza a classe `Conta1`.

```
1 class Banco1 {
2   Conta1 a = new Conta1(100.00);
3   Conta1 b = new Conta1(200.00);
4 }
```

A criação de um objeto pela operação `new Banco1()` produz a alocação de objetos da Fig. 2.10.

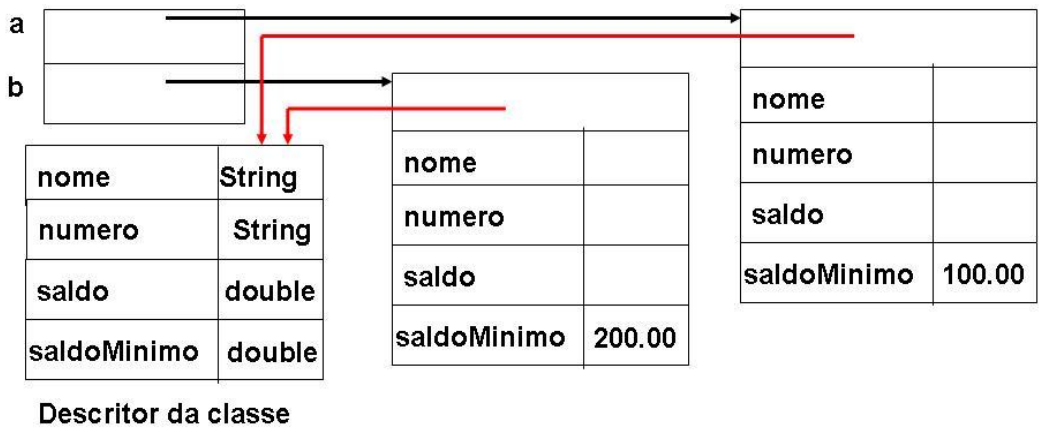


Figura 2.10 Duas contas do `Banco1`

Note que todo objeto do tipo `Conta1` tem seu próprio saldo mínimo. Em situações nas quais os valores de saldo mínimo variam de conta para conta, essa solução é adequada. Entretanto, se o saldo mínimo tiver que ser o mesmo para todas as contas, seria

um desperdício ter seu valor armazenado em cada uma das contas. Bastaria armazenar um único valor, que seria compartilhado por todas as demais contas. Além disto, bastaria mudar o valor compartilhado para que o efeito chegasse a todas as contas.

Variáveis de classe são o recurso de Java para prover esse tipo de compartilhamento. Variáveis de classe são campos de classe declarados com o modificador `static`.

O exemplo a seguir mostra como variáveis de classe podem ser declaradas.

```
1 class Conta2 {
2    private String nome;
3    private String numero;
4    private double saldo;
5    public static double saldoMinimo = 100.00;
6 }
```

onde `saldoMinimo` é uma variável de classe. As demais são variáveis de instância. O leiaute de objetos do tipo `Conta2` tem a forma da Fig. 2.11.

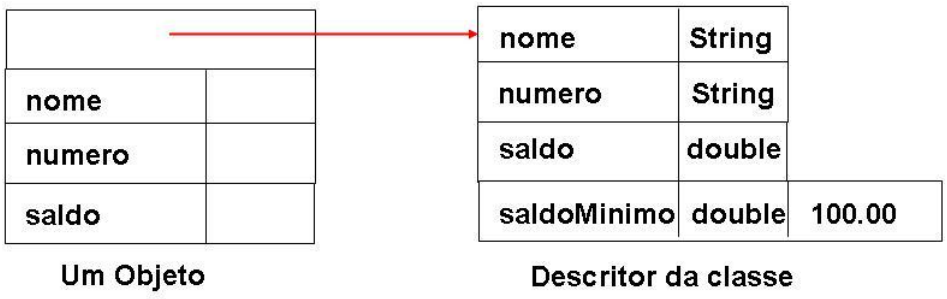


Figura 2.11 Leiaute com `static`

Note que o campo estático `saldoMinimo` é alocado no próprio descritor da classe e não na área dos objetos que forem instanciados. Campos estáticos são automaticamente iniciados com valores *default*, na primeira vez em que a classe que os contém for encontrada durante a execução, e seus valores podem ser alterados a

qualquer momento.

No programa **Banco2** a seguir, exemplifica-se o caso em que todas as contas devam ter sempre ter o mesmo limite de saldo mínimo, e que qualquer alteração desse limite deve aplicar-se a todas as contas.

```
1 class Banco2 {
2    Conta2 a = new Conta2();
3    Conta2 b = new Conta2();
4 }
```

E a execução da operação de criação de objeto `new Banco2()` produz a Fig. 2.12.

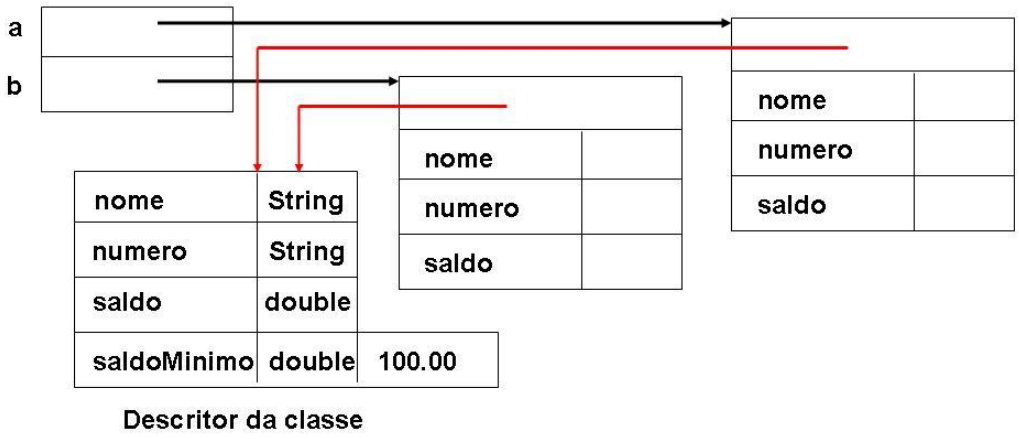


Figura 2.12 Conta com saldo mínimo `static`

Embora variáveis de classe ou campos estáticos não pertençam ao leiaute dos objetos, eles podem ser tratados como se a eles pertencessem. Em particular, no programa **Banco2**, a partir das referências **a** e **b** do tipo **Conta2**, pode-se ter acesso direto ao campo **saldoMinimo**, ou, então, pode-se fazê-lo pela sua qualificação pelo nome da classe, como mostrado abaixo:

```
Conta2.saldoMinimo = 1000.00;  // saldo mínimo == 1000.00
a.saldoMinimo += 1000.00;      // saldo mínimo == 2000.00
b.saldoMinimo -= 2000.00;      // saldo mínimo == 0.00
```

Um método que faz acesso a somente variáveis de classe pode ser declarado estático, por meio do modificador `static`, e, assim sendo, ele pode ser executado independentemente dos objetos instanciados para classe, como ocorre com `f` na linha 7 do exemplo a seguir.

```
1 class A {
2   static int x = 100;
3   static private int f() {return x;}
4 }
5 public class TesteDeA {
6    public static void main(String[] args) {
7       System.out.println("valor de f() = " + A.f());
8    }
9 }
```

Métodos estáticos não possuem o parâmetro implícito `this`, que foi discutido na Seção 2.7. Assim, `this` não pode ser usado em seu corpo, conforme mostra o programa abaixo, que obtém do compilador Java a mensagem `Undefined Variable this`, pela tentativa de usar ilegalmente a referência `this` na linha 3 do programa abaixo.

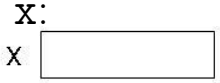
```
1 class B {
2    static int x = 100;
3    static private int f() { return this.x; }
4 }
5 public class TestedeB {
6   public static void main(String[] args) {
7      System.out.println("valor de f() = " + B.f());
8   }
9 }
```

2.9 Arranjos de objetos

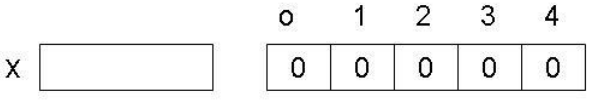
Arranjos de objetos são de fato arranjos de referências a objetos. Quando um arranjo desse tipo é criado, seus elementos são alocados no *heap* e automaticamente iniciados com o valor `null`. No curso do programa, os endereços dos objetos a que se referem os elementos do arranjo devem ser explicitamente atribuídos a cada um desses elementos. Para exemplificar esse processo, considere o programa `Arranjo`, que contém um arranjo de tipo primitivo `int` e um outro de referências a objetos.

```
1  public class Arranjo {
2     public static void main(String[] a) {
3        int[] x = new int[5];
4        x[2] = x[1] + 1;
5        A[] y = new A[5];
6        y[2] = new A(1,2);
7        y[5] = new A(4,6);
8     }
9  }
10 class A {
11     private int r, s;
12     public A(int d, int e) {r = d; s = e;}
13 }
```

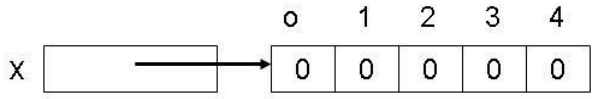
Quando o `Arranjo.main` for ativado, o efeito da declaração `int[] x`, da linha 3, é a alocação, na pilha de execução da JVM, de uma célula não iniciada `x`:



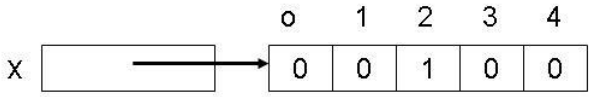
A seguir, a operação `new int[5]`, da linha 3, aloca no *heap* um arranjo de 5 inteiros, todos zerados, e devolve seu endereço:



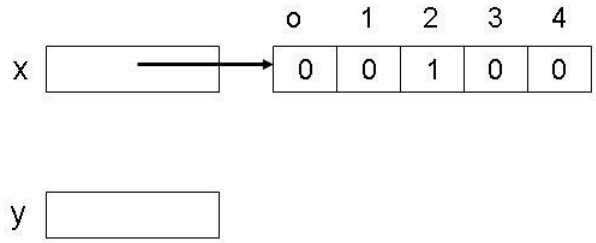
O último ato da atribuição da linha 3 inicia `x` com o endereço do arranjo criado, produzindo:



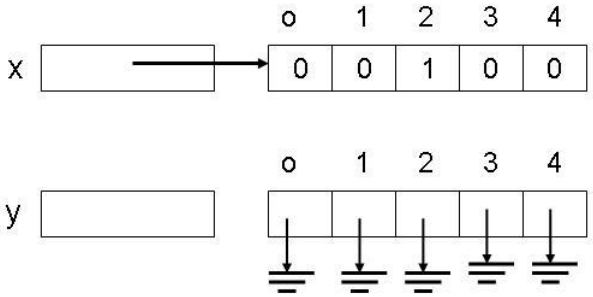
O comando de atribuição `x[2] = x[1] + 1`, da linha 4, atualiza o elemento de arranjo `x[2]` com o valor `1`:



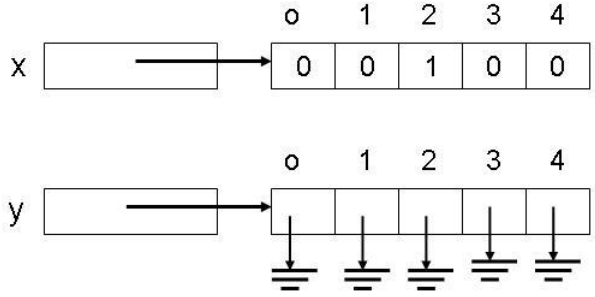
A declaração `A[] y`, da linha 5, aloca `y` na pilha de execução, produzindo:



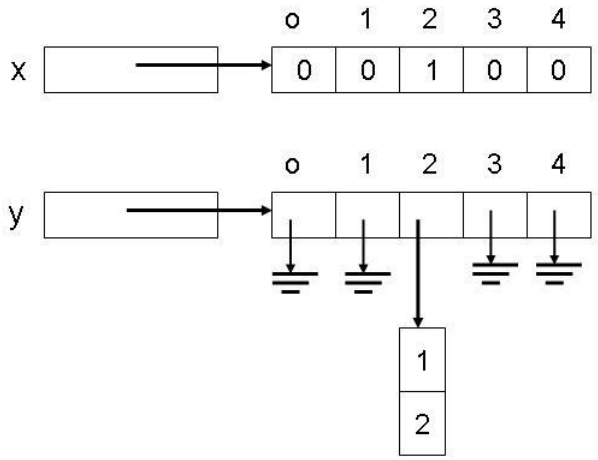
A operação `new A[5]`, da linha 5, aloca no *heap* um arranjo de cinco referências para objetos do tipo `A`. Os elementos desse arranjo são iniciados como o valor `null`:



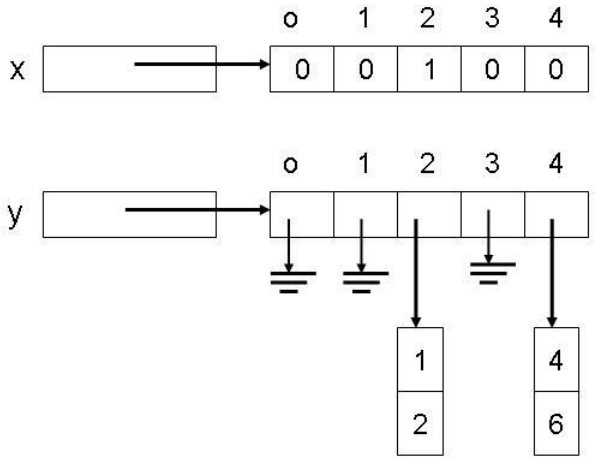
Após a execução do comando de atribuição da linha 5, tem-se a seguinte configuração de objetos:



O comando da linha 6 aloca um objeto do tipo `A` e atribui seu endereço à célula `y[2]`, produzindo:



Após a execução do último comando de `Arranjo.main`, o da linha 7, tem-se a configuração final dos objetos criados pelo programa `Arranjo`:



2.10 Iniciação de objetos

Variáveis de classe, de instância e componentes de arranjos são iniciados com valores *default*, conforme o tipo de cada um, no momento em que são criados.

Parâmetros, que são passados por valor, são iniciados com o valor dos argumentos correspondentes.

Variáveis locais a métodos não são iniciadas automaticamente,

mas devem ser explicitamente iniciadas pelo programador de uma forma verificável pelo compilador.

Em resumo, variáveis alocadas no *heap* são sempre iniciadas automaticamente, enquanto as alocadas na pilha de execução da JVM devem ser iniciadas explicitamente no programa.

Campos estáticos, ou variáveis de classe, são alocados e iniciados automaticamente quando a classe que os contém for usada pela primeira vez no programa para criar objetos ou para acessar algum de seus campos estáticos. A iniciação de variáveis de classes ocorre em três fases:

- no momento da carga da classe, com valores *default*, conforme o seu tipo
- com valores definidos pelos iniciadores contidos na declaração das variáveis estáticas da classe
- pela execução de blocos de iniciação de estáticos, definidos na classe.

Campos não-estáticos, ou campos de objetos, também chamados de variáveis de instância, são iniciados quando o objeto é criado. A iniciação desses campos ocorre segundo os passos:

- atribuição de valores *default* na criação do objeto
- atribuição dos valores indicados nos iniciadores na declaração
- execução de blocos de iniciação de não-estáticos
- execução de funções construtoras.

Uma iniciação simples normalmente é feita por *default* ou por meio de expressão de iniciação colocada junto à declaração do campo a ser iniciado. Para iniciações mais complexas, usam-se funções construtoras e blocos de iniciação. Os blocos de iniciação podem ser de estáticos e de não-estáticos.

Blocos de iniciação de estáticos são executados quando a classe acabou de ser carregada, mas logo após as iniciações *default* e

a execução das iniciações indicadas nas declarações dos campos estáticos. Blocos de iniciação de estáticos funcionam como se fosse uma *função construtora* dedicada exclusivamente a campos estáticos. Blocos de inicialização de não-estáticos são executados quando o objeto é criado, imediatamente antes de se executar o corpo da construtora, que ocorre depois das iniciações *default* e das contidas nas declarações das variáveis de instância.

Se uma classe tiver mais de um bloco de inicialização, eles serão executados em ordem de sua ocorrência, dentro de sua categoria, estáticos ou não-estáticos.

O programa a seguir mostra diversas formas de iniciação de campos não-estáticos:

```
1  public class IniciaNãoEstáticos {
2     int x = 10;
3     {x += 100; System.out.print(" x1 = " + x);}
4     public A() {
5        x += 1000; System.out.print(" x2 = " + x);
6     }
7     public A(int n) {
8        x += 10000; System.out.print(" x4 = " + x);
9     }
10    public static void main(String[] args) {
11       A a = new A() ;
12       A b = new A(1) ;
13    }
14 }
```

Note que, na linha **??**, há um iniciador de declaração, na linha 3, está um bloco de iniciação de não-estáticos, nas linhas 4 a 6, uma função construtora, e nas linhas 7 a 9, outra função construtora.

O objetivo do exemplo acima é mostrar os passos da execução de iniciação de objetos, que produz a saída:

`x1 = 110 x2 = 1110 x1 = 110 x4 = 10110`,

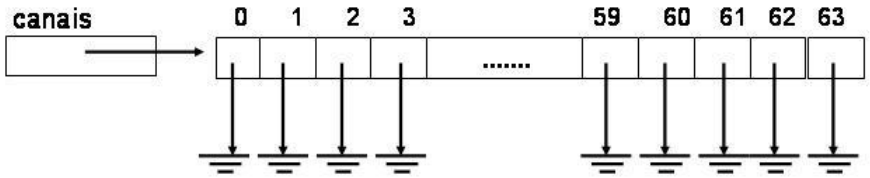
cuja análise revela detalhes do fluxo de execução, mostrado, a seguir, linha a linha.

A execução começa pela função `main`, definida na linha 10. Na linha 11, inicia-se a criação de um objeto do tipo `A`, passando-se sequencialmente pelas linhas 4, 2 e 3. Nesse ponto, tem-se a saída `x1 = 110` e conclui-se a criação do objeto com a execução do corpo da construtora na linha 5, que produz a saída `x2 = 1110`. O fluxo de execução então volta à linha 12, de onde inicia-se a criação do segundo objeto do tipo `A`, passando pelas linhas 7, 2 e 3. Nesse ponto, produz-se a saída `x1 = 110`, e a criação do segundo objeto conclui-se com a execução da construtora da linha 8, que produz a saída `x4 = 10110`. A função `main` termina na linha 13.

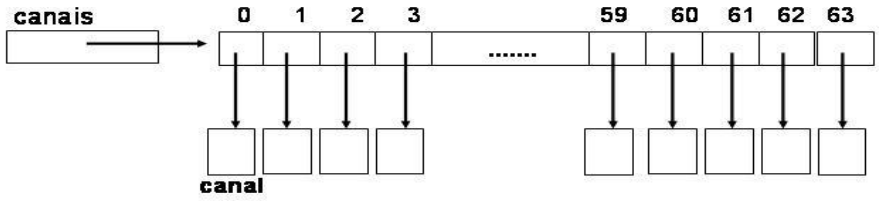
O exemplo abaixo exibe a necessidade de se usar blocos de iniciação de variáveis estáticas, porque as operações de iniciação necessárias são mais complexas:

```
1  public class IniciaEstáticos {
2     int t;
3     static final int MAXCANAIS = 64;
4     static Canal canais[] = new Canal[MAXCANAIS];
5
6     static {
7        for (int i = 0; i < MAXCANAIS; i ++) {
8            canais[i] = new Canal(i);
9        }
10    }
11 }
12 class Canal {public Canal(int ch) { ... } }
```

Quando o descritor da classe `IniciaEstáticos` for carregado, as suas variáveis de classe são alocadas e iniciadas. Após o processamento do iniciador de arranjo `canais` da linha 4, tem-se:



A seguir, a execução do bloco de iniciação de estáticos contido nas linhas 6 a 10 produz:



completando a iniciação da variável de classe `canais`.

2.11 Alocação de variáveis de classe

Variáveis de instância são alocadas e iniciadas no momento da criação do objeto. Variáveis de classe são alocadas e iniciadas no momento em que sua classe for referenciada pela primeira vez e seu descritor for carregado.

As classes `X, A, B, C` e `D`, definidas a seguir, são usadas pelo programa `LoadClass5`, que ilustra o mecanismo de alocação.

A classe `X` é usada para imprimir marcas para revelar o caminho percorrido pelo programa `LaodClass5`, definido a seguir.

```
1 class X {
2   public static int f(char m) {
3      System.out.print(" " + m); return 1;
4   }
5 }
```

Definem-se abaixo quatro classes quase idênticas, todas contendo diversos tipos de inicialização.

```
1  class A {
2     public static int r = X.f('A'), v = X.f('A');
3     static {r = 100; System.out.print(" r = " + r);}
4     {r = 1000; System.out.print(" r = " + r);}
5     public A() {r = 10000; System.out.print(" r = " + r);}
6  }
7  class B {
8     public static int s = X.f('B'),  w = X.f('B');
9     static {s = 100; System.out.print(" s = " + s);}
10    {s = 1000; System.out.print(" s = " + s);}
11    public B() {s = 10000; System.out.print(" s = " + s);}
12 }
13 class C {
14    public static int t = X.f('C'), z = X.f('C');
15    static {t = 100; System.out.print(" t = " + t);}
16    {t = 1000; System.out.print(" t = " + t);}
17    public C() {t = 10000; System.out.print(" t = " + t);}
18 }
19 class D {
20    public static int u = X.f('D'), k = X.f('D');
21    static {u = 100; System.out.print(" u = " + u);}
22    {u = 1000; System.out.print(" u = " + u);}
23    public D() {u = 10000; System.out.print(" u = " + u);}
24 }
```

O programa `LoadClass5` abaixo, por meio de marcações, mostra o momento e a ordem de execução dos eventos.

Sua execução inicia-se com a criação de um objeto do tipo `A` na linha 4, causando além da alocação de espaço para objeto, a iniciação de `r` e `v`, a execução do bloco de estático, do bloco de não-estático e da função construtora de `A`, resultando na saída:

```
P1 A A r = 100 r = 1000 r = 10000
```

Observe que na linha 5, apenas espaço para a referência `b` é alocado na pilha de execução. Nenhuma outra alocação é feita.

Assim, apenas a marca `P2` é impressa. Note que o mesmo acontece na execução da linha 7.

```
1  public class LoadClass5 {
2     public static void main(String[] args) {
3       PrintStream o = System.out;
4       o.println("P1");   A a = new A();
5       o.println("P2");   B b;
6       o.println("P3");   A c = new A();
7       o.println("P4");   B d;
8       o.println("P5");   int x = B.s;
9       o.println("P6");   int y = B.s;
10      o.println("P7");   B e = new B();
11      o.println("P8");
12         if (args.length==0) {C g = new C();}
13         else {D g = new D();}
14      o.println("Acabou");
15    }
16 }
```

Na linha 6 outro objeto do tipo `A` é criado, mas a iniciação dos campos estáticos `r` e `v` não é mais realizada, e também não se executa mais o bloco de iniciação de estáticos da classe. A saída após a marca `P3` mostra que apenas o bloco de não-estáticos e a função construtora foram executados:

```
P3 r = 1000 r = 10000
```

O uso do campo estático `s` da classe `B` na linha 8 força a carga da classe `B`, com a alocação e iniciação de seus campos estáticos `s` e `w` e da execução do seu bloco de iniciação de estáticos:

```
P5 B B s = 100
```

Na linha 9, nenhuma alocação ou iniciação é necessária, pois a classe `B` já foi alocada na linha anterior. Apenas a marca `P6` é impressa.

Quando se cria um objeto do tipo `B` na linha 10, se `B` já estiver alocada, e, portanto, também já estarão seus campos estáticos.

Assim, nesse momento, aloca-se espaço para o objeto, iniciam-se os campos não-estáticos, executam-se o bloco de não-estáticos e a função construtora, produzindo a saída:

```
P7 s = 1000 s = 10000
```

Se o programa foi disparado sem parâmetros, `args.length` tem valor `0`, e a saída produzida pela linha 13 seria

```
P8 C C t = 100 t = 1000 t = 10000
```

mostrando que nada da classe `D` foi alocado.

A saída total do programa `LoadClass5` é:

```
P1 A A r = 100 r = 1000 r = 10000
P2
P3 r = 1000 r = 10000
P4
P5 B B s = 100
P6
P7 s = 1000 s = 10000
P8 C C t = 100 t = 1000 t = 10000
Acabou
```

A pequena aplicação apresentada a seguir tem o propósito de mostra a utilidade de campos e funções estáticos.

2.12 Uma pequena aplicação

O problema que se deseja resolver é mostrar a evolução do saldo de uma conta de poupança, na qual faz-se um depósito mensal fixo, durante um período de 10 anos.

A conta remunera mensalmente o capital nela acumulado com base em uma taxa de juros anuais fornecida, e essa taxa deve ser a mesma para todas as contas.

A conta de poupança é modelada como um tipo abstrato de dados pela classe **Conta** apresentada a seguir, onde os membros de dados são mantidos privados e somente as operações que caracterizam o tipo são declaradas públicas.

```
1  public class Conta {
2     static private double taxaAnual;
3     private double saldoCorrente;
4     public static double taxa() {return taxaAnual;}
5     public static void defTaxa(double valor) {
6        if (valor > 0.0 && valor < 12.0) taxaAnual = valor;
7     }
8     public void creditar(double q) {
9        if (q > 0.0) saldoCorrente += q;
10    }
11    public void debitar(double q) {
12       if (q>0.0 && q<=saldoCorrente) saldoCorrente -= q;
13    }
14    public double saldo() { return saldoCorrente;}
15 }
```

Observe que o fato de todas as contas serem remuneradas pela mesma taxa de juros foi modelada por meio do membro **taxaAnual**, linha 2, que foi declarado como uma variável de classe, e os demais atributos são declarados como variáveis de instância.

O programa faz algumas operações de consistência nos dados, como taxa de juros, que devem estar dentro de certos limites, e que os valores de depósitos devem ser positivos. Para simplificar o exemplo, operações com dados inconsistentes são ignoradas.

Para exibir o desempenho da conta mediante depósitos regulares e rendimentos creditados, o programa **Poupança** apresentado a seguir lê da linha de comando o valor do depósito mensal e a taxa anual de juros, calcula e imprime a evolução dos saldos ao longo de 10 anos.

```
1  public class Poupança {
2    public static void main(String args[]) {
3      double deposito = (double)Integer.parseInt(args[0]);
4      double taxaAnual = (double)Integer.parseInt(args[1]);
5      Conta c = new Conta();
6      double juros;
7      Conta.defTaxa(taxaAnual);
8      System.out.println("Depositando " + deposito +
9                "por mes a taxa de " + Conta.taxa());
10     System.out.println("Veja a Evolução:");
11     for (int ano = 0 ; ano <10; ano++) {
12        System.out.print(ano + " - ");
13        for (int mes = 0; mes < 12; mes++) {
14          juros = c.saldo()*Conta.taxa()/1200.0;
15          c.creditar(juros); c.creditar(deposito);
16          System.out.print(c.saldo() + ", ");
17        }
18        System.out.println();
19     }
20   }
21 }
```

2.13 Conclusão

O conceito de classe introduzido neste capítulo é o ponto central que caracteriza uma linguagem orientada por objetos. É a partir desse conceito que são construídos os mecanismos de hierarquia de tipos, polimorfismo, tipos estáticos e dinâmicos e tipos abstratos de dados, que fundamentam uma linguagem de programação moderna.

Exercícios

1. Qual é a relação entre classes e tipos abstratos de dados?

2. Quando é que uma classe define um tipo abstrato de dados? E quando apenas define um tipo?

3. Para que servem blocos de iniciadores de estáticos? Funções construtoras não seriam suficientes para produzir os resultados oferecidos por esses blocos?

4. Blocos de iniciação de membros de dados não-estáticos são de fato necessários? O que se perderia sem eles?

Notas bibliográficas

As páginas da Oracle (http://www.oracle.com) ao lado dos livros publicados pelos criadores da linguagem Java [1, 2] são as mais importantes referências bibliográficas para essa linguagem.

O livro dos Deitel [10] é uma boa referência para aqueles que vêem Java pela primeira vez. Embora seja um pouco verboso, esse livro apresenta muitos exemplos esclarecedores do funcionamento da linguagem Java.

Capítulo 3

Interfaces

Tipos classificam dados, provendo uma organização nos valores usados em um programa. A separação de dados em tipos, como inteiros, fracionários, cadeias de caracteres e booleanos, permite implementação eficiente das operações de cada tipo, além de facilitar a detecção de erros decorrentes de mistura indevida de valores. Todo tipo tem uma interface que deve ser claramente especificada. Para isso, em Java há a construção `interface`, que é um recurso linguístico para especificar as assinaturas das operações de tipos abstratos de dados a ser implementados por meio da declaração de classes.

3.1 Declaração de interfaces

Uma interface consiste na especificação de cabeçalhos de métodos abstratos, definições completas de métodos estáticos e de métodos `default` e de declarações de constantes simbólicas, sendo estas consideradas como campos estáticos. A visibilidade de qualquer membro de interface é por *default* sempre `public`.

Em uma interface, métodos abstratos não podem ter seu corpo definido, mas os estáticos e os *defaults* devem sempre declarados de forma completa, com seus corpos. Além disto, interfaces valorizam a independência de implementação, por isso, métodos de interface

não podem ter modificadores de características de implementação, como `native`, `synchronized` ou `strictfp`.

Não se pode criar objetos a partir de uma interface, uma vez que somente os cabeçalhos de certos métodos são definidos, e a representação dos objetos nunca está definida na interface. Objetos devem ser criados a partir de classes concretas que implementam a sua interface. Classes concretas devem declarar os membros de dados que forem necessários e dar corpos aos métodos da interface. Mais de uma classe pode implementar uma mesma interface, e objetos dessas classes têm o mesmo tipo, que é o da interface.

3.2 Implementação de interfaces

Para ilustrar as vantagens do uso de interfaces no lugar de classes concretas, considere as declarações a seguir.

```
1  public class Bolo {
2     private double calorias, gorduras;
3     public Bolo(double c, double g) {
4        calorias = c; gorduras = g;
5     }
6     public String identidade() {return "Bolo" ;}
7     public double calorias() {return this.calorias;}
8     private double gorduras() {return this.gorduras;}
9  }
10 public class Torta {
11    private double calorias, gorduras;
12    public Bolo(double c, double g) {
13       calorias = c; gorduras = g;
14    }
15    public String identidade() {return this.calorias;}
16    private double gorduras() {return this.gorduras;}
17 }
```

A classe `Usuário1`, apresentada a seguir, que faz uso de `Bolo` e `Torta`, possui dois métodos, `g1` e `g2`, que fazem exatamente a mesma tarefa, diferindo apenas nos tipos dos objetos que aceitam como parâmetros. Isso exibe uma situação de baixa reusabilidade de código, manifestada pela duplicação dos corpos desses métodos.

Duplicação de código é uma prática considerada nociva à manutenção do sistema e deveria sempre ser evitada. O ideal é que se tivesse apenas um método que fosse capaz de executar indistintamente a tarefa desejada sobre os dois tipos de objetos, `Bolo` e `Torta`, manipulados pelo programa. Interface é uma solução que minimiza esse problema de duplicação desnecessária de código.

```
1  public class Usuário1 {
2    private double g1(Bolo  y) {return y.calorias();}
3    private double g2(Torta y) {return y.calorias();}
4    public void main() {
5      PrintStream o = System.out;
6      Bolo   b = new Bolo(150.0, 19.0);
7      Torta  t = new Torta(300.0, 56.0);
8      double z = g1(b) + g2(t);
9      o.println(b.identidade() + ":");
10     o.println("  " + b.calorias() + " kcal");
11     o.println("  " + b.gorduras() + "g de gorduras");
12     o.println(t.identidade() + ":");
13     o.println("  " + t.calorias() + " kcal");
14     o.println("  " + t.gorduras() + "g de gorduras");
15     o.println("Total de Calorias de = " + z);
16   }
17 }
```

O primeiro passo de modificação do programa acima para aumentar o grau de reúso de seus componentes consiste na observação de que as classes `Bolo` e `Torta` apresentam-se para a `Usuário1` como tendo exatamente as mesmas operações, i.e., os métodos `identidade()`, `calorias()` e `gorduras()`. Isso sugere que se

coloquem os tipos dos objetos `Bolo` e `Torta` na mesma família definida por uma interface comum, por exemplo, `Comestível`.

```
1 public interface Comestível {
2    double calorias();
3    String identidade();
4    double gorduras();
5 }
```

A interface `Comestível` especifica as operações de um tipo, que pode ser implementado por meio de classes como `Bolo` e `Comestível`, que iniciam a formação de uma hierarquia de tipos centrada em `Comestível`. A Fig. 3.1 exibe essa hierarquia.

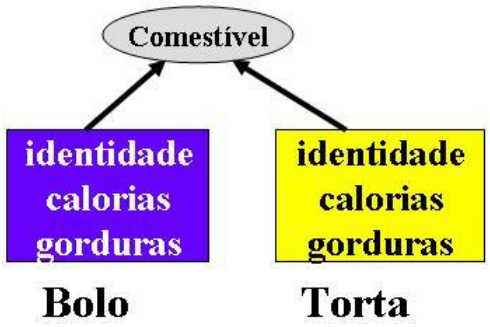


Figura 3.1 Hierarquia de `Comestível`

A obrigação de uma classe que implementa uma interface é dar corpos aos métodos listados na interface, como a seguir:

```
1 public class Bolo implements Comestível {
2    private double calorias, gorduras;
3    public Bolo(double c, double g) {
4       calorias = c; gorduras = g;
5    }
6    public String identidade() {return "Bolo";}
7    public double calorias() {return calorias;}
8    public double gorduras() {return gorduras;}
9 }
```

Uma declaração da forma `class A implements I` estabelece um relacionamento **é-um** entre a classe `A` e a interface `I`. Esse relacionamento autoriza o uso de objetos do tipo `A` onde objetos do tipo `I` forem esperados. Outro exemplo é a classe `Torta`:

```
1  public class Torta implements Comestível {
2     private double calorias;
3     private double gorduras;
4     public Torta(double c, double g) {
5        calorias = c; gorduras = g;
6     }
7     public String identidade() {return "Torta";}
8     public double calorias() {return calorias;}
9     public double gorduras() {return gorduras;}
10 }
```

O estabelecimento da relação **é-um** definida na hierarquia exibida acima coloca `Bolo` e `Torta` na mesma família, e assim sugere a fusão dos métodos `g1` e `g2` de `Usuário1` para produzir o método `g`, que aparece na versão `Usuário2` abaixo.

```
1  public class Usuário2 {
2    private double g (Comestivel y) {
3        return y.calorias();
4    }
5    public void  main(String [] args) {
6      PrintStream o = System.out;
7      Comestível b = new Bolo(150.0, 19.0);
8      Comestível t = new Torta(300.0, 56.0);
9      double z = g(b) + g(t);
10     o.println(b.identidade()+":"+b.calorias+","+b.gorduras);
11     o.println(t.identidade()+":"+t.calorias+","+t.gorduras);
12     o.println("Total de Calorias de = " + z);
13   }
14 }
```

Note que o método `g` de `Usuário2` aceita como parâmetro tanto referências a objetos do tipo `Bolo` como as do tipo `Torta`.

3.3 Compartilhamento de implementação

Uma mesma classe pode implementar várias interfaces e deve, no mínimo, prover o corpo de cada um dos métodos contidos na união de todos os cabeçalhos de métodos das interfaces implementadas.

Note que quando houver ocorrências de um mesmo cabeçalho de método em mais de uma interface a ser implementada por uma mesma classe, somente um corpo de método é necessário para implementar todas as ocorrências desse mesmo cabeçalho.

Por exemplo, considere as interfaces:

```
1  public interface Sobremesa {
2     String identidade();
3     double calorias();
4  }
5
6  public interface Comestível {
7     double calorias();
8     String identidade();
9     double gorduras();
10 }
```

Essas duas interfaces podem ser implementadas separadamente por classes distintas, ou então por uma única classe, como a classe `Fruta` abaixo, que implementa a união dos métodos declarados nas duas interfaces, onde dois métodos `identidade` e dois métodos `calorias`, herdados das interfaces, são implementados uma única vez cada um.

Ressalta-se que isto é possível porque os cabeçalhos dos dois métodos `identidade` e dos dois `calorias` são respectivamente

iguais. Se houvesse diferença nos parâmetros, cada um deveria ter sua própria implementação.

Por outro lado, se a diferença fosse no tipo retornado pelas funções, **Fruta** não poderia implementar simultaneamente ambas as interfaces.

```
1 public class Fruta implements Sobremesa, Comestível {
2    private double calorias, acidez;
3    public Fruta(double c, double a) {
4       calorias = c; acidez = a;
5    }
6    public String identidade() {return "Fruta";}
7    public double calorias() {return calorias;}
8    public double gorduras() {return 0.0;}
9 }
```

Uma outra implementação possível dessas duas interfaces é a classe **Bolo**, que concretiza a ideia de que um **Bolo** é **Sobremesa** e também é **Comestível**:

```
1 public class Bolo implements Sobremesa, Comestível {
2    private double calorias;
3    private double gorduras;
4    public Bolo(double c, double g) {
5       calorias = c; gorduras = g;
6    }
7    public String identidade() {return "Bolo" ;}
8    public double calorias() {return calorias ;}
9    public double gorduras() {return gorduras;}
10 }
```

As declarações de **Bolo** e **Fruta** acima criam a hierarquia de tipos da Fig. 3.2:

Observe que as implementações das operações da classe **Bolo**, a saber, **identidade**, na linha 7, e **calorias**, na linha 8, definem os corpos dos respectivos cabeçalhos de operações oriundas de ambas

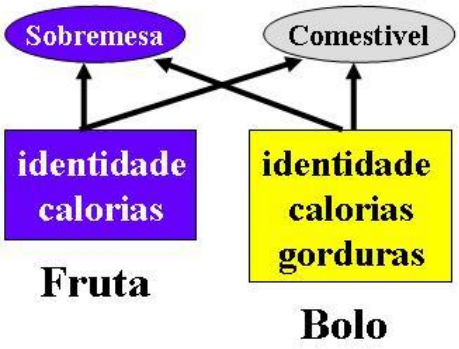


Figura 3.2 Hierarquia de `Sobremesa` e `Comestível`

as interfaces implementadas por `Bolo`, e que todo objeto do tipo `Bolo` pode ser usado onde um `Comestível` ou uma `Sobremesa` forem esperados.

O programa `Usuário3` mostra o uso das classes `Bolo` e `Torta` que implementam as interfaces `Sobremesa` e `Comestível` e destaca o polimorfismo das funções `f` e `g`.

```
1  class Usuário3 {
2     private String f(Sobremesa x) {x.calorias();}
3     private double g(Comestivel y) {
4        return y.gorduras();
5     }
6     public void testeUsuário () {
7        Fruta a  = new Fruta(50.0,3.0);
8        Bolo  b  = new Bolo(150.0,19.0);
9        double z = g(a) + g(b);
10       System.out.println(a.identidade() + " : " + f(a));
11       System.out.println(b.identidade() + " : " + f(b));
12       System.out.println("Total de gorduras " + z);
13    }
14 }
```

Uma classe pode implementar zero ou mais interfaces, mas somente pode estender no máximo **uma** outra classe. Se nenhuma extensão for especificada, a classe, por *default*, estende `Object`. Assim, os métodos de uma interface podem ser definidos direta-

mente nas classes que a implementam, ou então herdados de outras classes. O formato geral da declaração de uma classe é:

```
[public] class NomeDaClasse [extends Superclasse]
                [implements Interface1, Interface2, ...] {
     declarações de métodos e atributos

}
```

3.4 Hierarquia de interfaces

Pode-se construir hierarquia de interfaces, fazendo interfaces estenderem outras. Nesse processo, todos os protótipos e constantes definidos em uma interface tornam-se elementos das respectivas interfaces derivadas, as quais podem acrescentar outros elementos de interface. A construção de hierarquias de interfaces facilita o reúso de código e torna o programa mais legível.

Protótipos de métodos idênticos são unificados na interface derivada. Constantes simbólicas herdadas tornam-se elementos da interface derivada. Os casos de conflitos de nomes devem ser resolvidos via qualificação pelo nome da interface de origem.

A única situação de real conflito ocorre quando os protótipos de métodos herdados de diferentes interfaces diferem-se apenas no tipo de retorno. Essas situações devem ser evitadas, pois o tipo de retorno não é usado para distinguir chamadas de métodos, e, portanto, esses protótipos não podem ser implementados por um mesmo método, e não é possível escrever mais de um método cujos cabeçalhos se diferem apenas no tipo de dado a ser retornado. Ou mudam-se os nomes em conflito ou muda-se a hierarquia.

As interfaces que formam uma hierarquia de tipos podem ser individualmente implementadas, por meio de classes concretas independentes.

O exemplo de uma hierarquia simples é mostrada na Fig. 3.3, a qual define quatro interfaces por meio das seguintes declarações:

```
1 public interface Figura {
2    void desenha();
3    void apaga();
4    void desloca();
5 }
6 interface Polígono  extends Figura {double perímetro();}
7 interface Retângulo extends Polígono {double área();}
8 interface Hexágono  extends Polígono {double área();}
```

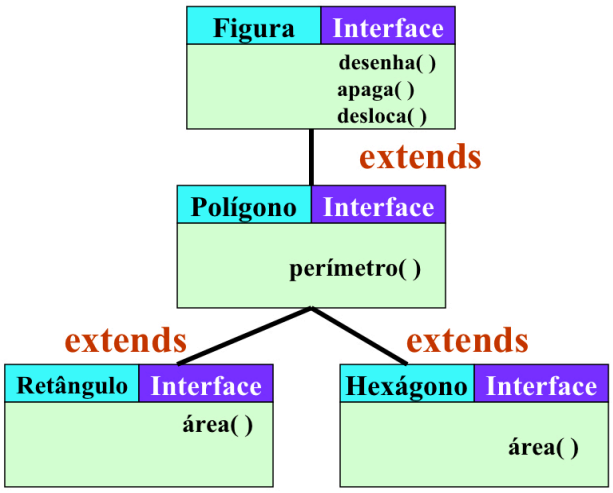


Figura 3.3 Hierarquia de Interfaces

As classes que implementam interfaces localizadas em quaisquer pontos na hierarquia também são implementações de todos os tipos que estejam nos caminhos que vão desde a interface implementada até as raízes da *árvore* de tipos[1].

O formato geral de definição de uma interface é:

```
[public] interface Nome [extends Interface1, ...] {
   campos estáticos, públicos e finais
   protótipos de métodos públicos
}
```

[1] A hieraquia pode não formar uma árvore no sentido estrito, por causa da possibilidade de se ter herança múltipla de interfaces

onde vê-se a possibilidade de se construir herança múltipla de interfaces. As classes `MeuRetângulo` e `SeuRetângulo`, definidas a seguir, mostram duas implementações diretas do tipo `Retângulo`.

```
1  class MeuRetângulo implements Retângulo {
2     "membros de dados"
3     public void desenha() {...}
4     public void apaga() { ... }
5     public void desloca() { ... }
6     public double perímetro() { ... }
7     public double área() { ... }
8  }
9  class SeuRetângulo implements Retângulo {
10    "membros de dados"
11    public void desenha() { ... }
12    public void apaga() { ... }
13    public void desloca() { ... }
14    public double perímetro() { ... }
15    public double área() { ... }
16 }
```

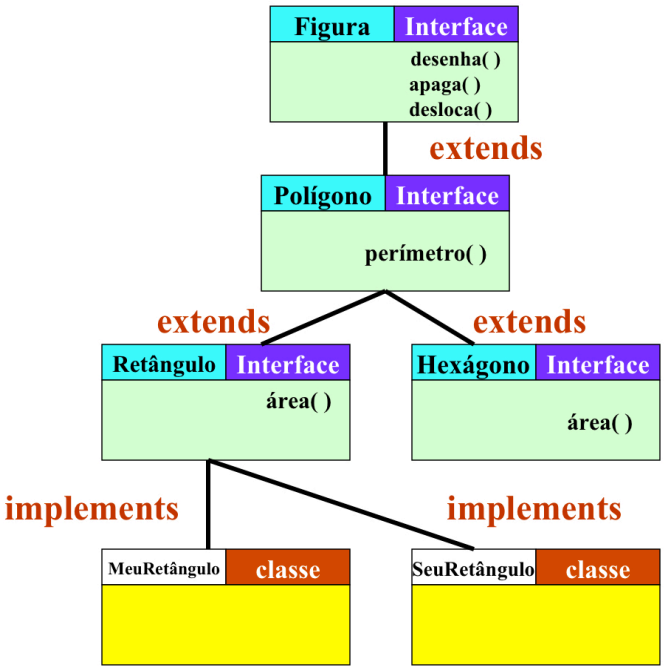


Figura 3.4 Hierarquia de Figuras

Na hierarquia de tipos da Fig. 3.4, as classes **MeuRetângulo** e **SeuRetângulo** implementam todas as interface localizadas no caminho até a raiz a árvore. Para destacar o polimorfismo de referência, considere a classe:

```
1  public class Polivalente {
2     void m1(Figura f) {...; f.desenha(); ... }
3     void m2(Polígono p) {
4        ...; p.desenha(); ...; p.perímetro();...
5     }
6     void m3(Retângulo r) {
7        ...; r.desenha(); ...; r.área(); ...
8     }
9     void m4(Hexágono r) {
10       ...; r.desenha(); ...; r.área(); ...
11    }
12    void m() {
13       Figura q = new MeuRetângulo();
14       m1(q);  // Correto
15       m2(q);  // Correto
16       m3(q);  // Correto
17       m4(q);  // Errado no tipo do parâmetro
18    }
19 }
```

A referência **q** declarada do tipo **Figura** na linha 13 da classe **Polivalente**, e feita apontar para objeto do tipo **MeuRetângulo**, é passada para os métodos **m1**, **m2** e **m3**, os quais esperam um parâmetro do tipo **Figura**, **Polígono** e **Retângulo**, respectivamente, ilustrando o elevado grau de reúso que se pode alcançar com o uso de interfaces. O mesmo ocorreria se a **q** fosse atribuído uma referência a objetos do tipo **SeuRetângulo**.

Observe, contudo, que, nos corpos das funções **m1**, **m2** e **m3**, são permitidos acessos somente aos membros do objeto a elas passados que pertençam ao tipo estático do parâmetro.

Note que o comando `m4(q)` na linha 17 da classe `Polivalente` é inválido, e assim será apontado como um erro de tipagem estática pelo compilador Java, porque o tipo `Hexágono` não foi declarado como um superior hierárquico de `MeuRetângulo`, ou seja, o tipo `MeuRetângulo` não é um tipo `Hexágono`, como é de se esperar.

3.4.1 Uso de interface

A representação ideal da estrutura interna de um tipo abstrato de dados depende da implementação da operação que se deseja dar maior eficiência, pois a escolha da representação dos dados influencia diretamente o desempenho, para mais ou para menos, das operações sobre a estrutura.

É prática usual privilegiar o desempenho das operações de uso mais frequente. Entretanto, há situações em que as representações possíveis produzem desempenhos contraditórios de operações igualmente frequentes, dificultando a escolha. Nesses casos, a melhor solução é implementar todas as representações, para que a mais conveniente em cada situação seja a usada.

Interface é um mecanismo capaz de prover, de forma transparente, esse tipo de implementação dual, que é ilustrada a seguir com a implementação do tipo abstrato de dados `Complexo`, o qual é definido pela seguinte interface:

```
1 public interface Complexo {
2     void some(Complexo z);
3     void subtraia(Complexo z);
4     void multiplique(Complexo z);
5     void divida(Complexo z);
6 }
```

As operações do tipo `Complexo`, apesar de exibirem apenas um operando, são, de fato, binárias, sendo o primeiro operando o ob-

jeto receptor, o segundo, o argumento passado ao método, e o resultado fica armazenado no objeto receptor. Ou seja, `x.some(y)` é equivalente a `x = x + y`, para `x` e `y` do tipo `Complexo` em coordenadas polares e `+` representando a semântica de `some`.

Tradicionalmente, há duas representações para um número complexo: coordenadas cartesianas e coordenadas polares, conforme mostra a Fig. 3.5, onde o para (x,y) das coordenadas cartesianas denota o mesmo valor do complexo (ρ,θ).

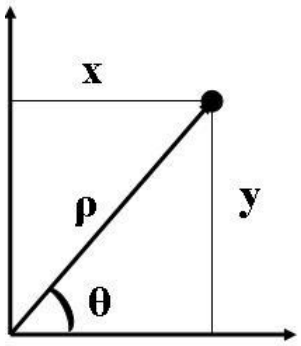


Figura 3.5 Representação do `Complexo x + yi`

Sabe-se que coordenadas cartesianas favorecem o desempenho das operações `some` e `subtraia`, enquanto coordenadas polares facilitam as operações `multiplique` e `divida`, conforme mostram as fórmulas abaixo:

- Coordenadas Cartesianas:
  - Soma: $(a + bi) + (c + di) = (a + c) + (b + d)i$
  - Subtração: $(a + bi) - (c + di) = (a - c) + (b - d)i$
  - Multiplicação: $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$
  - Divisão: $\frac{(a+bi)}{(c+di)} = \left(\frac{ac+bd}{c^2+d^2}\right) + \left(\frac{bc-ad}{c^2+d^2}\right) i$
- Coordenadas Polares:
  - Soma: $\rho_1 \, \mathrm{e}^{i\theta_1} + \rho_2 \, \mathrm{e}^{i\theta_2} = (\rho_1 * \cos\theta_1 + \rho_2 * \cos\theta_2) + i(\rho_1 * \sin\theta_1 + \rho_2 * \sin\theta_2)$

– Subtração: $\rho_1 \, \mathrm{e}^{i\theta_1} + \rho_2 \, \mathrm{e}^{i\theta_2} = (\rho_1 * \cos\theta_1 - \rho_2 * \cos\theta_2) + i(\rho_1 * \sin\theta_1 - \rho_2 * \sin\theta_2)$

– Multiplicação: $\rho_1 \, e^{i\theta_1} \cdot \rho_2 \, e^{i\theta_2} = \rho_1 \, \rho_2 \, e^{i(\theta_1+\theta_2)}$

– Divisão: $\frac{\rho_1 \, e^{i\theta_1}}{\rho_2 \, e^{i\theta_2}} = \frac{\rho_1}{\rho_2} \, e^{i(\theta_1-\theta_2)}$.

onde $\rho \, \mathrm{e}^{i\theta}$ representa o número complexo $\rho \, (\cos\theta + i \sin\theta)$.

Supondo que essas quatro operações sejam de uso igualmente frequente, sugere-se a implementação de ambas as representações por meio das classes concretas `Cartesiano` e `Polar`, de forma que cada operação possa ser realizada com a representação que lhe for mais apropriada. A classe `Dualidade` declarada a seguir mostra o convívio harmônico das duas representações para números complexos em um mesmo programa.

```
1  public class Dualidade {
2     public static void main(String args) {
3        Complexo a = new Cartesiano(2.0,2.0);
4        Complexo b = new Cartesiano(1.0,1.0);
5        Complexo c = new Polar();
6        Complexo d = new Polar(1.0, 2.0);
7        a.some(b);        // a = a + b
8        b.subtraia(c);    // b = b - c
9        c.divida(d);      // c = c / d
10       System.out.println("a = " + a.toString());
11       System.out.println("b = " + b.toString());
12       System.out.println("c = " + c.toString());
13       System.out.println("d = " + d.toString());
14    }
15 }
```

Os objetos `a` e `b` foram criados via a construtora de *Cartesiano*, enquanto `c` e `d`, via a de *Polar*. A ideia é que as operações do tipo `Complexo` façam automaticamente as conversões para o formato mais conveniente quando ativadas.

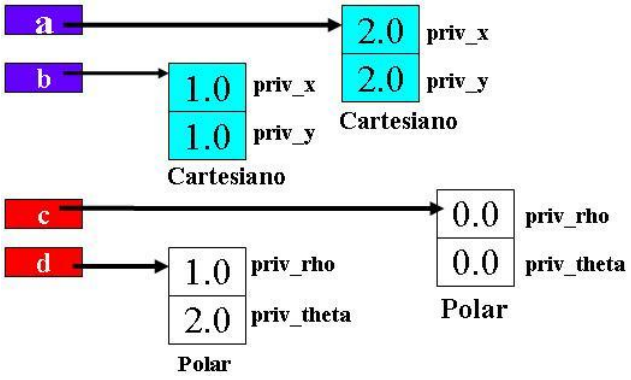


Figura 3.6 Leiaute dos Objetos de `Dualidade`

O leiaute dos objetos alocados pelo **`main`** de **`Dualidade`** é mostrado na Fig. 3.6. E a classe concreta **`Cartesiano`**, que representa o número complexo pelos membros **`privX`** e **`privY`**, é a seguinte:

```
1  public class Cartesiano implements Complexo {
2     private double privX, privY;
3     public Cartesiano(double x, double y) {
4        privX = x;  privY = y;
5     }
6     public Cartesiano() {this(0.0, 0.0);}
7     public double x() {return privX;}
8     public double y() {return privY;}
9     public static Cartesiano converta(Complexo w) {
10       Cartesiano c; Polar p;
11       if (w instanceof Polar) {
12           p = (Polar) w;  c = new Cartesiano();
13           c.privX = p.rho()*cos(p.theta());
14           c.privY = p.rho()*sin(p.theta());
15       } else c = (Cartesiano)w;
16       return c;
17    }
18    public void some(Complexo r) {
19       Cartesiano rc = Cartesiano.converta(r);
20       privX = privX + rc.privX;
21       privY = privY + rc.privY;
22    }
```

```
23     public void subtraia(Complexo r) {
24        Cartesiano rc = Cartesiano.converta(r);
25        privX = privX - rc.privX;
26        privY = privY - rc.privY;
27     }
28     public void multiplique(Complexo r) {
29        Polar p = Polar.converta(this);
30        p.multiplique(r);
31     }
32     public void divida(Complexo r) {
33        Polar p = Polar.converta(this);
34        p.divida(r);
35     }
36     public String toString() {
37        StringBuffer b = new StringBuffer();
38        b.append(privX).append(" + ").
39                    append(privY).append('i');
40        return b.toString();
41     }
42  }
```

As operações **some**, **subtraia**, **multiplique** e **divida** declaradas no tipo **Cartesiano** aceitam objetos do tipo **Complexo** como parâmetro, e, portanto, podem receber um objeto **Cartesiano** ou **Polar**.

As operações **some** e **subtraia** são mais eficientes se realizadas em coordenadas cartesianas. Para isso, usa-se o método **converta**, definido a partir da linha 9 da classe **Cartesiano**, para assegurar que o número complexo dado pelo parâmetro está devidamente representado em coordenadas cartesianas, e assim permitir que o cálculo do novo estado do objeto corrente possa ser feito em coordenadas cartesianas.

As operações de **multiplique** e **divida** das linhas 28 e 32 são realizadas com os operandos representados em coordenadas polares, porque, como mostrado acima, essas operações são mais facil-

mente implementadas nessa representação. O método **converta** declarado na linha 11 da classe **Polar**, definida a seguir, é usado para fazer as devidas conversões de formato.

```
1  public class Polar implements Complexo {
2     private double privRho, privTheta;
3     private final double PI = 3.14159;
4     public Polar(double rho, double theta) {
5        privRho = rho;  privTheta = theta;
6     }
7     public Polar() {this(0.0, 0.0);}
8     public double theta() {return privTheta;}
9     public double rho() { return privRho;}
10
11    public static Polar converta(Complexo w) {
12       Polar p ; Cartesiano c;
13       if (w instanceof Cartesiano) {
14           c = (Cartesiano) w; p = new Polar();
15           p.privRho = sqrt(c.x()*c.x()+c.y()*c.y());
16           p.privTheta = atan2(c.y(),c.x());
17       } else p = (Polar)w;
18       return p;
19    }
20    public String toString() {
21       Cartesiano c = Cartesiano.converta(this);
22       return c.toString();
23    }
24
25    public void some(Complexo r) {
26       Cartesiano c = Cartesiano.converta(this);
27       c.some(r);
28    }
29
30    public void subtraia(Complexo r) {
31       Cartesiano c = Cartesiano.converta(this);
32       c.subtraia(r);
33    }
```

```
34
35      public void multiplique(Complexo r) {
36          Polar rp = Polar.converta(r);
37          privRho = privRho * rp.privRho;
38          privTheta = mod(privTheta + rp.privTheta,2*PI);
39      }
40
41      public void divida(Complexo r) {
42          Polar rp = Polar.converta(r);
43          privRho = privRho/rp.privRho;
44          privTheta = mod(privTheta-rp.privTheta,2*PI);
45      }
```

A implementação acima da classe `Polar` representa números complexos em coordenadas polares, por meio dos campos privados `privRho` e `privTheta`.

Em conclusão, nessa implementação de números complexos, as operações `Polar.some` e `Polar.subtraia` são realizadas em coordenadas cartesianas, usando as correspondentes operações da classe `Cartesiano`. E as operações `multiplique` e `divida` da classe `Polar` são realizadas em coordenadas polares, com operações da classe `Polar`.

3.4.2 Interfaces e métodos estáticos

Interfaces não aceitam declarações de protótipos de métodos estáticos, porque esses protótipos exigem sempre um corpo definido junto aos seus cabeçalhos. Métodos estáticos em interfaces devem ser sempre declarados de forma completa com a especificação de seus corpos e não não há como redefini-los em subclasses ou subinterfaces.

Para exemplificar, a compilação do programa abaixo, que define a interface `I`, aponta um erro de compilação na linha 3 pela falta do corpo do método estático `f`.

```
1 public interface I {
2    static int x = 10;
3    static void f() ;  // Errado
4 }
```

Por outro lado, a declaração de `f` na linha 3 abaixo não guarda relação alguma com o cabeçalho do método homônimo declarado na interface `I`.

```
1 class B implements I {
2      private static int y;
3      public static void f() { y = 10;} // outro f
4 }
```

O programa a seguir exemplifica uma forma correta de implementar métodos estáticos em interfaces.

```
1 public interface C {
2    static int x = 100;
3    static void g() { x = 10;}
4 }
```

Em conclusão, métodos estáticos pertencem somente à interface na qual foram declarados, assim somente podem ser qualificados pelo nome da interface, e.g., `C.g()`, e nunca por classes que implementem a interface onde foram declarados.

3.4.3 Interfaces e métodos `default`

Após a disponibilização de uma interface, não se deve alterar o conjunto de operações por ela definido, sob pena de invalidar os programas que a utilizam. Entretanto, muitas vezes faz-se necessário acrescentar novas operações numa interface, mas isso pode sair muito caro, devido ao efeito nocivo de forçar todas as classes que já implementam essa interface a ser alteradas para dar os devidos

corpos a essas novas operações, mesmo não venham a usá-las. Essa atualização de código é necessária, pois é obrigação de toda classe definir os métodos abstratos herdados de uma interface ou classe abstrata.

Para resolver esse problema, Java 8 introduziu o conceito de métodos *default*, que são métodos definidos em interface de forma completa, i.e., com cabeçalho e corpo, e que podem ser redefinidos.

Métodos *default* são normalmente herdados por subinterfaces e subclasses, e, por que já têm corpos definidos, não demandam ser reimplementados nas classes que implementam a sua interface. E assim, podem ser livremente incorporados a interfaces já disponibilizadas, sem o risco de gerar atualizações nos programas anteriores.

Os programas `E1` e `E2`, a seguir, ilustram a aplicação desse recurso.

```
1  interface I {
2     int fator = 10;
3     int compute(int n);
4  }
5  public class E1 implements I {
6     public int compute(int n) { return fator * n; }
7     public static void main(String args[]) {
8        I1 x = new E1();
9        int y = x.compute(20);
10       System.out.println("Resultado = " + y);
11    }
12 }
```

Suponha que a interface `I` declarada acima, depois de algum tempo após ser disponibilizada, deva ser alterada no sentido de incluir novas operações, preservando a sintaxe e a semântica da operações existentes.

E para que isso tenha efeito somente nas futuras implementações de `I`, os métodos devem ser incluídos como *default*, como mostrado

pelo programa abaixo.

```
1 interface I {
2    int fator = 10;
3    int compute (int n);
4    default int converte(int v) {return v * 2.54;}
5    }
6 }
```

Note que a alteração realizada na interface `I`, pela inclusão de um método *default* `converte`, não tem impacto no programa `E1` acima, que permanece inalterado e correto, pois ele não precisa implementar o método `converte`, que agora também é herdado.

Por outro lado, pode-se agora construir uma classe `E2`, que implementa a nova `I`, e faz uso do método `converte` na linha 7.

```
1 public class E2 implements I {
2    public int compute(int n) { return fator * n; }
3    public static void main(String args[]) {
4       I1 x = new E1();
5       int y = x.compute(20);
6       int z = x.converte(y);
7       System.out.println("Resultado = " + z);
8    }
9 }
```

Os métodos *default* foram incorporados em Java, a partir da versão 8, para permitir que classes como `E1` e `E2`, que implementam diferentes versões de uma mesma interface, convivam harmoniosamente em um mesmo sistema de computação. Há contudo um sinal de alerta no uso de métodos *default*: a introdução desses métodos em interfaces e o fato de classes poderem implementar múltiplas interfaces podem gerar ambiguidade de referência quando um mesmo método é herdado de mais de uma interface, como ilustra o seguinte programa, no qual supõe-se uma situação

em que haja duas interfaces `I1` e `I2`, que especificam as mesmas operações.

```
1  interface I1 {
2     int fator = 10;
3     int compute (int n);
4     default int converte(int v) { return v * 2.54;}
5     }
6  }
7  interface I2 {
8     int fator = 10;
9     int compute (int n);
10    default int converte(int v) { return v * 0.39;}
11    }
12 }
```

E suponha que a classe `E3` tente, sem sucesso, implementar ambas as interfaces `I1` e `I2`, da seguinte forma:

```
1  public class E3 implements I1, I2 {
2     public int compute(int n) { return fator * n; }
3
4     public static void main(String args[]) {
5        I1 x = new E1();
6        int y = x.compute(20);
7        int z = x.converte(y);
8        System.out.println("Resultado = " + z);
9     }
10 }
```

O insucesso dessa tentativa é revelado pelo erro de compilação no comando da linha 7, pois, como `E2` implementa `I1` e`I2`, há nesse ponto do programa duas funções `converte` disponíveis.

A solução desse problema seria definir localmente uma nova função `converte` que use uma das funções herdadas devidamente qualificada pelo nome de sua respectiva interface, no caso `I1` ou `I2`, conforme sugerido na linha 4 do programa:

```
1  public class E4 implements I1, I2 {
2     public int compute(int n) { return fator * n; }
3     public int converte(int v) {
4        I1.super.converte(v); // ou I2.super.convert(v);
5     }
6     public static void main(String args[]) {
7        I1 x = new E1();
8        int y = x.compute(20);
9        int z = x.converte(y);
10       System.out.println("Resultado = " + z);
11    }
12 }
```

3.5 Aninhamento em Interfaces

Classes podem ser aninhadas em interfaces com o objetivo de melhor agrupar os componentes de um programa, facilitando sua manutenção. O princípio fundamental é sempre manter juntos componentes que estão associados e tentar abstrair-se o máximo possível das partes do programa que tratam de outros temas.

Para ilustrar o valor dessa metodologia, considere a implementação de dois sistemas completamente independentes. Um desses sistemas destina-se a suportar avaliação de qualidade de vinhos, e o outro, para controle de frequência de alunos a aulas.

3.5.1 Solução elementar

Os itens da carta de vinhos são definidos pela classe `Item`.

```
class Item {
   public String rótulo;
   public int safra;
   public int nota;
}
```

A interface `Carta` especifica as operações do tipo `Vinho`:

```
interface Carta {
   void insere(Item x);
   Item retire(String rótulo);
   Item pesquise(String rótulo);
   void imprima();
}
```

O tipo `Vinho`, que implementa `Carta`, pode ser:

```
class Vinho implements Carta {
   private int tamanho;
   private Item[] itens;
   public Vinho(int tamanho) {
      this.tamanho = tamanho;
      itens = new Item[tamanho];
   }
   public void insere(Item x) { ... }
   public Item retire(String rótulo) { ...}
   public Item pesquise(String rótulo) { ... }
   public void imprima() { ... }
}
```

E o programa principal `Main1` tem a seguinte estrutura:

```
public class Main1 {
   public static void main(String args[]) {
      Vinho vinhos = new Vinho(10);
      Item v = new Item();
      v.rótulo = "merlot"; v.safra = 1910; v.nota = 9;
      vinhos.insere(v); ... vinhos.imprima(); ...
   }
}
```

O segundo sistema, o controle de frequência foi escrito à semelhança do sistema de controle de vinho, com o propósito de facilitar a comparação de soluções e no qual os itens do controle de frequência são do tipo:

```
class Item {
   public String nome;
   public int faltas;
}
```

A interface **Frequência** define as operações do tipo **Controle**:

```
interface Frequência {
   void insere(Item x);
   Item anotaPresença(String nome);
   void imprima();
}
```

E **Controle** que implementa a interface **Frequência** é a seguinte:

```
class Controle implements Frequência {
   private int tamanho;
   private Item[] itens;
   public Vinho(int tamanho) {
      this.tamanho = tamanho;  itens = new Item[tamanho];
   }
   public void insere(Item x) { ... }
   public Item anotaPresença(String nome) { ...}
   public Item imprima() { ... }
}
```

E o programa principal agora é **Main2**:

```
public class Main2 {
   public static void main(String args[]) {
      Controle alunos = new Vinho(10);
      Item a = new Item();
      a.nome = "maria"; a.faltas = 0;
      alunos.insere(a); ... alunos.imprima(); ...
   }
}
```

Os dois sistemas são colocados em operação e funcionam a contento. Mais tarde, esses sistemas devem ser incorporados a um

mesmo novo ambiente para dar o devido suporte computacional, por exemplo, a um curso sobre vinhos, onde há dois tipos de itens: um relativo a alunos e outro a vinhos. Essa unificação irá apresentar problemas, porque o nome `Item` passa a designar, no mesmo escopo, dois tipos distintos, e isso não é permitido em Java.

A solução imediata seria trocar o nome de um deles, tomando-se o cuidado de garantir a consistência dessa troca em todo o código, inclusive verificando se o novo nome é de fato inédito em todo o novo sistema. Dependendo da complexidade de código, troca de nomes pode ser muito trabalhosa e altamente sujeita a erros.

Por outro lado, toda essa dificuldade poderia ser evitada se o princípio anunciado no início desta seção tivesse sido observado, como mostra a solução detalhada a seguir.

3.5.2 Solução modular

Uma das fichas de dados denominada `Item`, usada em um dos exemplos em discussão, é uma informação vinculada ao tipo `Carta`, e por isso, deve pertencer a esse tipo, o que é feito pelo seu aninhamento na interface que a usa.

```
1  interface Carta {
2     class Item {
3        public String rótulo;
4        public int safra, nota.
5     }
6     void insere(Item x);
7     Item retire(int rótulo);
8     Item pesquise(int rótulo);
9     void imprima();
10 }
```

Esse aninhamento força a qualificação de `Item` pelo nome da interface que o contém, i.e., `Carta.Item`. Fora da hierarquia de

`Carta`, referências a `Item` devem ser escritas `Carta.Item`. E o programa `Main1` passa a ser:

```
1 public class Main1 {
2    public static void main(String args[]) {
3       Vinho vinhos = new Vinho(10);
4       Carta.Item v = new Carta.Item();
5       v.rótulo = "merlot"; v.safra = 1910; v.nota = 9;
6       vinhos.insere(v); ... vinhos.imprima(); ...
7    }
8 }
```

O mesmo pode ser feito com a interface `Frequência` do segundo sistema, que na solução dita modular, passa a ter o seguinte código:

```
1 interface Frequência {
2    class Item {
3      public String nome;
4      public int faltas;
5    }
6    void insere(Item x);
7    Item anotaPresença(String nome);
8    void imprima();
9 }
```

E o programa principal `Main2` passa a ter o código:

```
1 public class Main2 {
2    public static void main(String args[]) {
3       Controle alunos = new Vinho(10);
4       Frequência.Item a = new Frequência.Item();
5       a.nome = "maria"; a.faltas = 0;
6       vinhos.insere(a); ... alunos.imprima(); ...
7    }
8 }
```

A classe `Controle`, por estar na hieraquia de `Frequência`, fica inalterada e, finalmente, o programa `CursoDeVinho`, que aproveita

as implementações acima, poderia ter o segunte código, no qual as fichas de dados `Item` de `Carta` e de `Frequência` convivem harmoniosamente:

```
1  public class CursoDeVinho {
2     public static void main(String args[]) {
3        Controle alunos = new Vinho(10);
4        Frequência.Item a = new Frequência.Item();
5        Vinho vinhos = new Vinho(10);
6        Carta.Item v = new Carta.Item();
7        v.rótulo = "merlot"; v.safra = 1910; v.nota = 9;
8        vinhos.insere(v);
9        ... vinhos.imprima();   ...
10       a.nome = "maria"; a.faltas = 0;
11       vinhos.insere(a);
12       ... alunos.imprima(); ....
13 }
```

O grande benefício da encapsulação é permitir uniões sem traumas de códigos, muito necessárias quando trabalha-se com várias equipes. O controle de visibilidade de nomes ao longo de um programa é fundamental para se ter código mais modular e reusável.

3.6 Conclusão

Interfaces de Java é um rico mecanismo para atingir alto grau de reúso de código e deve ser bem explorado para a produção de programas modulares. A restrição de Java de somente permitir herança simples de classes simplifica a implementação de seu compilador e a possível perda de poder de expressão decorrente pode ser aliviada pelo uso de hierarquia de interfaces.

Exercícios

1. Por que métodos estáticos ou finais e também métodos com visibilidade `private` não podem ser declarados abstratos?
2. Como seria a tabela de métodos virtuais se existisse herança múltipla de classes em Java?
3. Qual seria a perda de poder de expressão se Java não tivesse o mecanismo de interfaces?
4. Qual a diferença entre uma interface e uma classe abstrata sem qualquer campo e somente com os mesmos protótipos da interface?

Notas bibliográficas

O conceito de hierarquia de classes é bem discutido em praticamente todos os textos sobre Java, e em particular no livro de K. Arnold, J. Gosling e D. Holmes [2, 3].

Capítulo 4

Hierarquia

Classes podem ser organizadas de forma a constituírem famílias que compartilham propriedades e operações. A partir dos conjuntos de operações comuns a duas ou mais classes pode-se formar uma hierarquia de classes, onde cada membro descreve as operações nele definidas, e que são também operações aplicáveis a todos os seus descendentes. Assim, um objeto cuja classe participa de uma hierarquia pode ser usado onde seja esperado um objeto de classe igual ou acima da sua nessa hierarquia.

A implementação de uma classe requer a declaração completa de seus atributos e dos corpos de seus métodos. Isto pode ser feito a partir do zero, definindo e implementando cada um dos membros de dados e métodos necessários, ou então, por meio de reúso de implementação, especializando classe existente e dela herdando declarações de atributos e implementações de métodos.

Nesse processo de reúso, pode-se, dentro de certos limites, adaptar os elementos herdados e, até, principalmente, acrescentar outros com o fim de prover novas funcionalidades.

Considerando que toda classe concreta implementa um novo tipo de dado, o processo de especialização de uma classe permite construir hierarquia de tipos, na qual todo tipo é um substituto dos que ficam acima dele na hierarquia. Os tipos posicionados acima de um dado tipo na hierarquia são chamados supertipos deste tipo,

e os tipos situados abaixo, subtipos.

Java oferece dois mecanismos para se criar hierarquia de tipos ou de classes: implementação de interfaces e extensão de classes e interfaces, que são descritos a seguir.

4.1 Hierarquia de classes

Ao declarar uma nova classe, pode-se aproveitar a declaração de membros de dados e de membros funcionais de outra classe anteriormente definida. Esse processo de reúso da implementação de uma classe na definição de outra chama-se herança de classe.

A classe criada é chamada de classe derivada ou subclasse, e a classe da qual se herda é dita classe base ou superclasse . A classe derivada herda todas as características de sua superclasse e pode adicionar outras.

Assim, herança de classe é um mecanismo para estender a funcionalidade de uma aplicação por meio da adição de outras funcionalidades. Diz-se que a classe derivada cria uma extensão de sua superclasse.

A Fig. 4.1 ilustra a construção das classes `Professor` e `Aluno` como subclasses ou extensão da classe `Pessoa`, que descreve a parte comum de objetos do tipo `Aluno` e `Professor`.

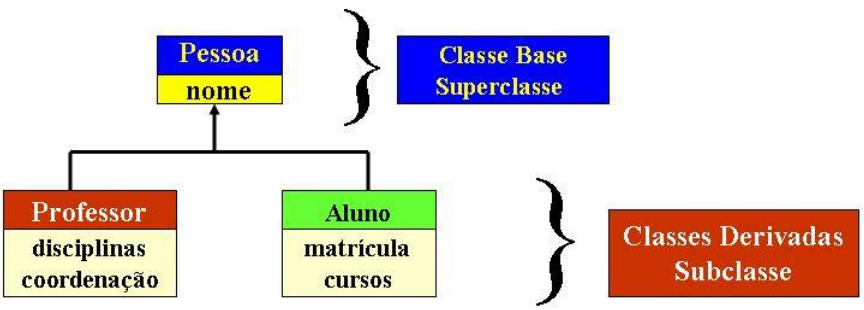


Figura 4.1 Hierarquia de Pessoas

Herança cria uma estrutura de subtipagem, na qual objetos

de um subtipo podem ser substitutos de objetos cujos tipos são seus supertipos. Isso ocorre porque os supertipos têm uma parte que é comum a todos os seus subtipos. Por exemplo, objetos do tipo `Pessoa` possuem o atributo `nome`, objetos da classe derivada `Professor` têm os campos `nome`, `disciplinas` e `coordenação`, enquanto objetos do tipo `Aluno`, também derivada de `Pessoa`, têm `nome`, `matrícula` e `cursos`.

Note que a parte `nome` da representação dos objetos `Professor` e `Aluno` possui a mesma estrutura da parte correspondente de objetos `Pessoa`. Isso permite que objetos do tipo `Professor` ou `Aluno` possam ser apresentados onde um objeto do tipo `Pessoa` é esperado, haja vista que toda a informação que forma um objeto desse tipo está disponível nos objetos do tipo `Professor` e `Aluno`. Portanto, `Professor` é uma `Pessoa`, bem como `Aluno` também é uma `Pessoa`. Genericamente, herança cria um relacionamento **é-um** entre subclasse e sua respectiva superclasse.

Uma implementação da hierarquia de classes da da Fig. 4.1 pode ser a seguinte:

```
1  class Pessoa {
2      private String nome;
3      outros membros (não exibidos na figura)
4  }
5  class Professor extends Pessoa {
6      private String disciplinas;
7      outros membros (não exibidos na figura)
8  }
9  class Aluno extends Pessoa {
10     private String matrícula;
11     private String cursos;
12     outros membros (não exibidos na figura)
13 }
```

Herança pode ser vista como um mecanismo que implementa

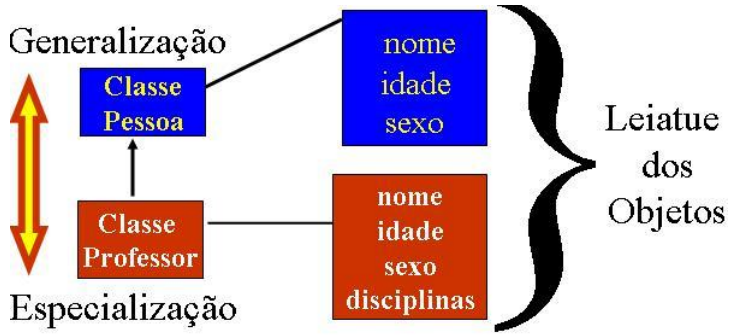


Figura 4.2 Especialização e Generalização

relações de **especializacão** e **generalização** entre uma classe e sua respectiva derivada, conforme ilustrada na Fig. 4.2. E não há limites no número de níveis hierárquicos.

A Fig. 4.3 mostra uma hierarquia em que a classe `Empregado` foi introduzida para encapsular os elementos comuns de objetos `Professor` e `Funcionário`. Nessa estrutura tem-se `Professor` e `Funcionário` como um `Empregado`, que, por sua vez, é uma `Pessoa`.

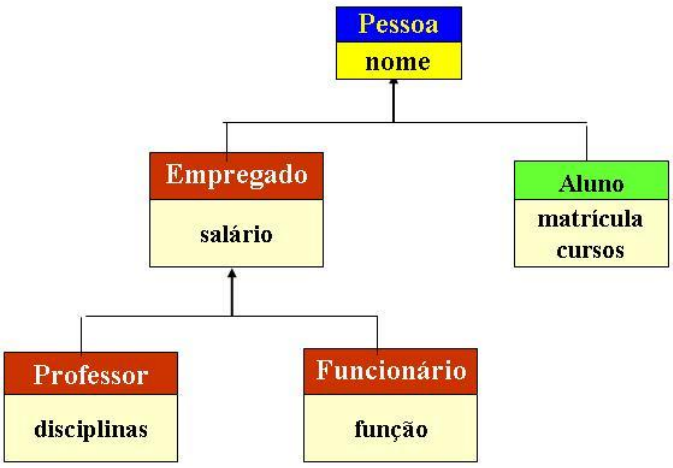


Figura 4.3 Hierarquia de Empregados

4.1.1 Reúso de implementação

Como foi dito, herança é um recurso para criar hierarquia de tipos por meio de um mecanismo para reúso de implementação, i.e., he-

rança permite aproveitar as declarações de atributos e de métodos de uma classe existente para se construir uma nova classe. Entretanto, esse aproveitamento de código deve ser feito sem violar a privacidade dos elementos das classes envolvidas.

Para fins de analisar os recursos oferecidos pelo mecanismo de herança, considere a classe `Hora`, que define hora com a precisão de segundos, e tem uma operação, `display`, para exibir, na saída padrão, a hora registrada no objeto corrente.

```
1 class Hora {
2    private int h, m, s;
3    public Hora(int hr, int min, int seg) {
4       h = hr; m = min; s = seg;
5    }
6    public void display() {
7       System.out.print(h + ":" + m + ":" + s);
8    }
9 }
```

Considere agora a classe `HoraLocal` que define um horário com especificação do local geográfico.

```
1 class HoraLocal {
2    private int h, m, s;
3    private String local;
4    public HoraLocal(int hr,int min,int seg,String local) {
5       h = hr; m = min; s = seg;
6       this.local = local;
7    }
8    public  void display() {
9       System.out.print
10           (h + ":" + m + ":"  +  s + " " +  local);
11   }
12 }
```

O programa **Horários** abaixo usa as classes **Hora** e **HoraLocal** para criar objetos, exibe os horários anotados nesses objetos e produz a saída: `11:30:50 10:26:10 bsb`.

```
1 public class Horários {
2    public static void main(String args) {
3       Hora t1 = new Hora(11,30,50);
4       HoraLocal t2 = new HoraLocal(10,26,10,"bsb");
5       t1.display();
6       System.out.print(" ");
7       t2.display();
8    }
9 }
```

Entretanto, observa-se que **Hora** e **HoraLocal** têm muito em comum, e por isso na implementação acima há uma repetição de código, que deveria ser eliminada. Para isso, a seguir, apresenta-se uma nova implementação da classe **HoralLocal**, que usa o mecanismo de herança para adaptar a implementação de **Hora** na criação da nova classe.

```
1 class HoraLocal extends Hora {
2    private String local;
3    public HoraLocal(int hr,int min,int seg,String local) {
4       super(hr,min,seg);
5       this.local = local;
6    }
7    public void display() {
8       super.display();
9       System.out.print(" " + local);
10   }
11 }
```

A nova classe **HoraLocal** possui os membros de dados **h**, **m** e **s**, herdados de **Hora**, e o membro **local**, que foi declarado diretamente na classe. Note que nem sempre pode-se ter acesso

direto a campos herdados. Por exemplo, o fato de `h`, `m` e `s` terem sido declarados `private` impede que `HoralLocal` lhes tenha acesso direto. Em particular, não se pode iniciá-los diretamente no corpo da classe `HoraLocal`. Sua iniciação deve ser via chamada à construtora da superclasse, que é feita, no caso, via o comando `super(hr,min,seg)` na linha 4. Dessa forma, a parte `Hora` de `HoraLocal` é devidamente iniciada. À construtora de `HoraLocal` compete iniciar diretamente apenas os campos locais declarados.

Para exibir a hora local, que depende dos campos privados `h`, `m` e `s` de `Hora`, usa-se o comando `super.display()`, na linha 8, que causa a execução do método `display` declarado dentro da superclasse `Hora`.

A palavra-chave `super` somente pode ser usada para denotar invocação do construtor da superclasse ou de métodos da superclasse. Seu valor é a referência ao objeto corrente, mas seu tipo dinâmico é a superclasse.

4.1.2 Classes abstratas

Interfaces de Java é um recurso de programação que permite o desenvolvimento de programas com um alto grau de abstração. Os usuários de uma interface apenas conhecem as assinaturas dos métodos e constantes por ela definidos e podem abstrair-se totalmente dos seus detalhes de implementação. Nada é fixado em relação à definição dos serviços oferecidos pelas classes que implementam a interface. Contudo, há situações em que se deseja fixar a definição de certos elementos de uma interface de forma a que sejam compartilhados por todas as classes que a implementam, enquanto outros somente têm suas assinaturas fixadas.

A solução para essa flexibilidade de expressão do nível de abstração desejado é o conceito de classe abstrata.

Classes abstratas são identificadas pela palavra-chave `abstract`. Nela define-se um conjunto de membros comuns às suas subclasses e as assinaturas de métodos a ser definidos mais tarde. Classes abstratas têm a mesma estrutura de uma classe normal, exceto que podem possuir um ou mais métodos sem corpo, ditos métodos abstratos, sendo assim definições incompletas de classes.

Se uma classe definida como uma implementação de uma interface deixar de implementar qualquer um dos métodos da interface, então a classe também é considerada abstrata e deve ser marcada com a palavra-chave `abstract`.

Classes abstratas podem ter construtoras, mas não se pode criar objetos diretamente a partir delas. Essas construtoras somente são usadas quando ativadas a partir de construtoras de subclasses. Normalmente as construtoras de classes abstratas deve ser visíveis em suas subclasses.

Métodos abstratos sempre devem ser definidos em alguma das subclasses da hierarquia que se pode formar estendendo a classe abstrata ou seus descendentes. As subclasses de uma classe abstrata em que todos os métodos herdados estejam devidamente implementados são chamadas de classes concretas.

Para ilustrar uma aplicação de classes abstratas, considere a situação em que se deseja construir uma forma geométrica sobre a qual sabe-se apenas que deve ter um posicionamento, definido por coordenadas cartesianas, e uma cor. Deseja-se também que as áreas de formas específicas, a ser definidas por subclasses, possam ser calculadas.

Como não se sabe ainda que formas concretas serão definidas pelos usuários, o máximo que se pode inferir a respeito das operações de cálculo de áreas são suas assinaturas, assim o método `área` não pode ter ainda corpo, como mostra a classe abstrata `Forma`:

```
1  import java.awt.*;
2  public abstract class Forma {
3     protected Color cor;
4     protected int x, y;
5     protected Forma(Color cor, int x, int y) {
6        this.cor = cor; this.x = x; this.y = y;
7     }
8     public abstract double área();
9     public abstract void desenha();
10 }
```

Note que os membros de **Forma** foram declarados **protected** ou **public** para tornar possível a futura implementação dos seus métodos abstratos. A partir da classe **Forma** pode-se, por exemplo, definir as classes concretas **Círculo** e **Retângulo**:

```
1  public class Círculo extends Forma {
2     private double raio;
3     public Círculo(Color c,int x,int y,double raio) {
4        super(c, x, y); this.raio =  raio;
5     }
6     public Círculo() { this(Color.red, 0, 0, 1);}
7     public double área () {return 3.1416*raio*raio;}
8     public void desenha() { ... }
9  }
10 public class Retângulo extends Forma {
11    private double largura;
12    private double altura;
13    public Retângulo
14              (Color c, int x, int y, double a, double b) {
15       super(c, x, y); this.largura = a; this.altura = b;
16    }
17    public void área() {return largura*altura;}
18    public void desenha() { ... }
19 }
```

O programa **Gráficos**, definido a seguir, usa a classe abstrata **Forma** para definir o tipo estático das referências **s1**, **s2** e **s3**, e as classes concretas **Círculo** e **Retângulo** para criar os objetos desejados. Observe que, a partir das referências definidas, as operações particulares de cada um dos objetos são corretamente ativadas.

```
1  public class Gráficos {
2    public static void main (String args) {
3      Forma s1 = new Retângulo(Color.red,0,5,200,300);
4      Forma s2 = new Círculo(Color.green,20,30,100);
5      Forma s3 = new Círculo();
6      Double área = s1.área() + s2.área() + s3.área();
7      s1.desenha();  s2.desenha();  s3.desenha();
8      System.out.println("Área total = " + área);
9    }
10 }
```

Fig. 4.4 mostra os objetos alocados pelo programa **Gráficos**.

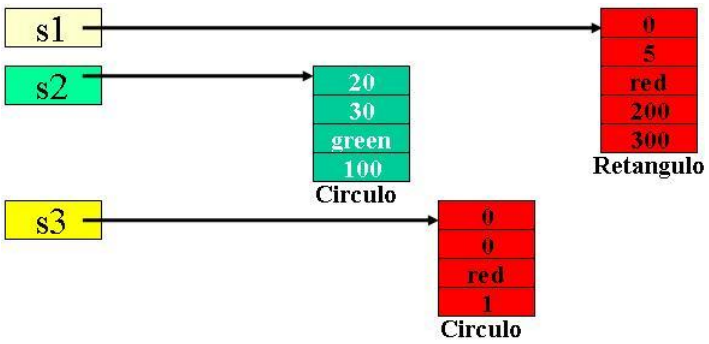


Figura 4.4 Objetos de Gráficos

4.2 Tipo estático e tipo dinâmico

Todo campo, variável ou parâmetro declarados do tipo referência a um objeto têm efetivamente, durante a execução, dois tipos a eles associados: tipo estático e tipo dinâmico.

O tipo estático é o tipo com o qual a referência foi declarada, e serve para informar que operações são aplicáveis ao objeto, e o tipo dinâmico é o tipo do objeto que em um dado momento a referência aponta.

Assim, o tipo estático de uma dada referência, como o nome indica, permanece o mesmo durante toda a execução do programa, enquanto seu tipo dinâmico pode mudar a cada atribuição ou passagem de parâmetro ocorrida no programa.

Dito de outra forma, o tipo estático define as mensagens que podem ser enviadas ao objeto a partir de sua referência, isto é, o tipo estático de uma referência informa os campos e métodos acessíveis a partir dela, os quais são os membros públicos declarados na classe dessa referência.

Por outro lado, o tipo dinâmico determina qual implementação do método associado a uma dada mensagem enviada ao objeto via a referência dada será executado, ou seja, a versão do método executada é sempre a definida na classe do objeto corrente denotado pela referência. O tipo dinâmico denota o tipo real do objeto a cada instante e, portanto, é ele que designa a classe do objeto que executa a ação. Como o tipo dinâmico de uma referência pode variar durante a execução de um programa, referências são ditas polimórficas. Note que, em Java, objetos são sempre monomórficos, sempre imutáveis, uma vez criados nunca mudam de forma.

O que pode ser polimórfico são as referências, pois uma mesma referência pode denotar objetos de classes distintas em diferentes momentos durante a execução, desde que estas classes pertençam a hierarquia do tipo estático associado.

As referências `this` e `super` são especiais. Seus tipos estáticos e dinâmicos são, respectivamente, iguais. Para a referência `this`, eles são sempre o tipo do objeto corrente. E os tipos estático e

dinâmico da referência **super** são a superclasse da classe onde a referência é usada.

Para ilustrar o papel da natureza dos tipos de uma referência durante a execução de um programa, considere a classe **Ponto** e a classe **Pixel**, derivada de **Ponto** para introduzir cor a cada ponto.

Os objetos das classes **Ponto** e **Pixel** abaixo têm os leiautes mostrados na Fig. 4.5.

```
1  class Ponto {
2     private double x, y;
3     public Ponto (double a, double b) {x = a; y = b;}
4     public void clear() {x = 0.0; y = 0.0;}
5     public double area() {return 0.0;}
6  }
7
8  class Pixel extends Ponto {
9     private Color color;
10    public Pixel (double a, double b, Color c) {
11       super(a,b);
12       color = c;
13    }
14    public Pixel () {this(0,0, Color.white);}
15    public void clear() {
16       this.color = null;
17       super.clear();
18    }
19    public void setcolor(Color c) {color = c;}
20 }
```

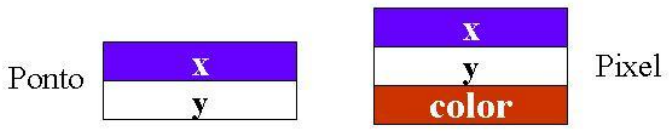


Figura 4.5 Leiaute de Objetos `Ponto` e `Pixel`

O programa **Janela** a seguir, no qual **p1** e **p2** são declarados com tipo estático **Ponto**, e **pixel**, com o tipo **Pixel**, ilustra como o tipo

dinâmico de um objeto muda durante a execução e o efeito que isso tem na seleção do método executado para as mesmas mensagens enviadas a `p1` e `p2` em diferentes pontos do programa.

```
1  public class Janela {
2      static public void main(String[] a) {
3         Ponto p1 = new Ponto(1,2);
4         Ponto p2 = new Pixel(3,4,Color.red);
5         Pixel pixel = new Pixel();
6         p1.clear();
7         p2.clear();
8         p1 = pixel;
9         p1.clear();
10     }
11 }
```

Após a execução dos comandos nas linhas 3, 4 e 5, os tipos dinâmicos de `p1`, `p2` e `pixel` são `Ponto`, `Pixel` e `Pixel`, respectivamente, e os objetos devidamente inicializados pelas suas construtoras são os mostrados na Fig. 4.6.

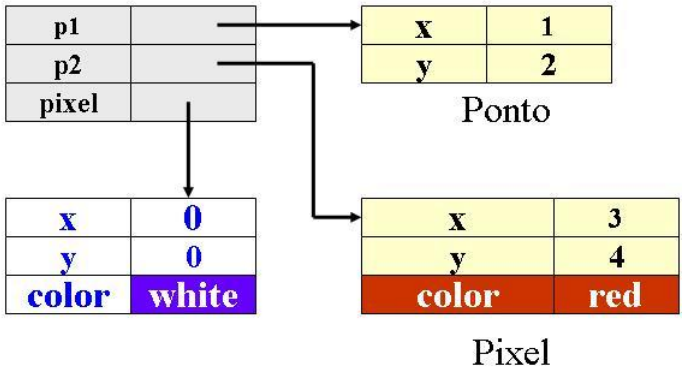


Figura 4.6 Configuração antes da linha 6

A execução do comando `p1.clear()` da linha 6 aciona o método `clear` da classe `Ponto`, e `p2.clear()` da linha 7 causa a execução do `clear` de `Pixel`, alterando o estado dos objetos `p1` e `p2` para configuração da Fig. 4.7.

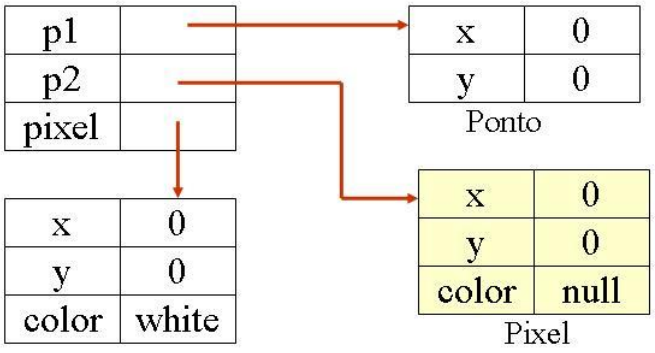


Figura 4.7 Configuração antes da linha 8

4.2.1 Atribuição de referências

O polimorfismo das referências a objetos, geradas pelo dinamismo do seu tipo, concede certa flexibilidade na atribuição ou passagem de parâmetros que são referências, a qual pode ser formalizada pela seguinte regra:

> *Uma referência para uma classe* `A` *pode ter o endereço de qualquer objeto de classe* `A` *ou de qualquer uma de suas subclasses.*

Suponha que uma classe `B` tenha sido declarada como uma subclasse de `A` e que as referências `a` e `b` sejam declaradas da forma:

```
A a = new A();
B b = new B();
```

De acordo com a regra acima, é válido o comando `a = b` ou a chamada `p(b)`, onde `p` tem a assinatura `T p(A a)`, para qualquer `T`. Essas operações fazem o tipo dinâmico de `a` ser `B` até que outra atribuição a `a` ocorra. Entretanto, não se permite a atribuição `b = a`, ou passar a referência `a` para um parâmetro do tipo `B`. Por razões de segurança, o compilador, nesses casos, exige que sempre se escreva `b = (B)a` e `p((B)a)`, para que o código de inspeção do tipo dinâmico de `a` seja devidamente gerado, e a atribuição da referência seja feita somente quando o tipo dinâmico de `a` for

de fato `B` ou uma subclasse de `B`. Caso esse não seja o caso, o programa tem uma exceção pertinente levantada. Boas práticas de programação recomendam testar o tipo do objeto, via operador `instanceOf`, antes de realizar a conversão.

O polimorfismo das referências também se aplica a arranjos, mas cuidado adicional deve ser observado. A regra básica é que, se `v` tiver tipo estático `A[]`, onde `A` é uma classe qualquer, então `v` pode legitimamente apontar para instâncias de qualquer arranjo de elementos do tipo `B`, desde que B seja uma subclasse de `A`.

Entretanto, esse compartilhamento dos elementos do arranjo deve ser feito cuidadosamente, pois não se pode alterar o tipo dos objetos apontados por elementos do arranjo `B[]` para tipos que não sejam subtipos de `B`.

Por exemplo, considere o programa `Atribuição`, onde o tipo estático de `v` é do tipo referência para arranjo de `A`, e o de `r`, referência para arranjo de `B`, sendo `B` uma subclasse de `A`.

```
1  class A {  }
2  class B extends A {int x;}
3  class Atribuição {
4     public static void main(String() args) {
5        A[] v = new A[5];
6        B[] r = new B[5];
7        for(int i=0;  i<5 ; i++) {
8            v[i] = new A();
9            r[i] = new B();
10       }
11       v = r;
12       v[0] = new A();   // <=== Erro de execução
13       r[0].x = 100;
14    }
15 }
```

A execução da função `main` da classe `Atribuição` até o ponto

imediatamente antes da linha 11 produz a alocação de objetos descrita na Fig. 4.8.

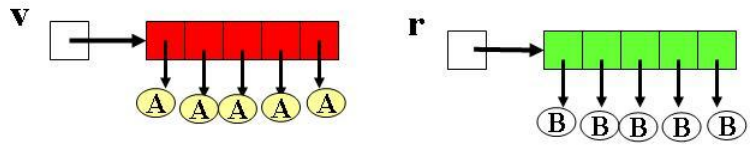


Figura 4.8 Vetores `v` e `r`

O comando na linha 11 produz a alocação da Fig. 4.9.

Figura 4.9 Vetores `v` e `r` após 11

A tentativa de execução da atribuição da linha 12 causa o lançamento da exceção `ArrayStoreException` e consequente cancelamento do programa. O lançamento dessa exceção visa impedir a situação mostrada na Fig. 4.10, que seria gerada se a atribuição fosse permitida, pois `r` deve referenciar um arranjo de `B`.

Figura 4.10 Vetores `v` e `r` após 12

Esses exemplos demonstram que nem sempre a regra de que uma referência para uma classe base `A` pode ter o endereço de qualquer objeto de classe `A` ou de qualquer uma de suas subclasses é válida, pois a questão do compartilhamento de objetos gerados pela atribuicão de referências deve ser considerado.

4.2.2 Inspeção de tipo dinâmico

O exemplo a seguir mostra o uso do operador `instanceof` para dar o tratamento seguro a referências polimórficas.

```
1  public interface Shape { }
2  public class Quadrado implements Forma {
3     private double lado;
4     public Quadrado(double lado) { this.lado = lado;}
5     public double lado() { return lado; }
6  }
7  public class Circulo implements Forma {
8      double raio;
9      public Circulo(double raio) { this.raio = raio; }
10     double raio() { return raio; }
11 }
12 public class Teste {
13    public static void main (String [ ]  args) {
14       Forma f;
15       Double area;
16       ...
17       f = new Circulo(10); ... f = new Quadrado(20); ...
18       ...  // nesse ponto qual é a Forma de f?
19       if (f instanceof Quadrado) {
20           Quadrado q = (Quadrado) f;
21           area =  2 * q.lado() * q.lado();
22       } else if (f instanceof Circulo) {
23             Circulo c = (Circulo) f;
24             area = Math.PI * c.raio() * c.raio();
25       } else {
26             throw new IllegalArgumentException("Forma?");
27       }
28       ...
29    }
30 }
```

Observe que as linhas 19 e 22 testam o tipo dinâmico da re-

ferência `f`, e que as linhas 20 e 23 fazem testes similares, pois essa é a semântica da operação de *casting*.

Para evitar essas repetições, Java SE 14 introduziu, na qualidade de *preview feature*, a possibilidade de integrar o teste do tipo dinâmico à atribuição automática do endereço do objeto a uma referência do tipo testado, como ilustra o exemplo a seguir.

```
1  public class Teste {
2     public static void main (String [ ]  args) {
3        Forma f;
4        Double area;
5        ...
6        f = new Circulo(10); ... f = new Quadrado(20); ...
7        ...  // nesse ponto qual é a Forma de f?
8        if (f instanceof Quadrado q) {
9            area =  2 * q.lado() * q.lado();
10       } else if (f instanceof Circulo c) {
11             area = Math.PI * c.raio() * c.raio();
12       } else {
13             throw new IllegalArgumentException("Forma?");
14       }
15       ...
16    }
17 }
```

Nessa *preview feature*, as variáveis `q` e `c` no exemplo acima são denominadas variáveis de ligação (*binding variables*) e somente podem ser usadas no trecho de programa que pode ser alcançado quando o resultado do `instanceof` para elas for `true`.

```
1 if (f instanceof Quadrado q) {
2        // nesse ponto o uso de q é ok
3 } else if (f instanceof Circulo c) {
4       // nesse ponto o uso de c é ok, mas q não
5 } else {// nesse ponto nem q nem c podem ser usados }
```

Essa regra de escopo pouco ortodoxa recomenda não usar o operador `instanceof` em operandos de expressões condicionais dotadas de curto-circuito, pois não se pode garantir sempre sua execução.

4.3 Execução de construtoras

O operador `new` usado para criação de objetos, inicialmente aloca uma área de memória de tamanho suficiente para conter os campos do objeto e os inicia com valores *defaults*, conforme o tipo de cada um. A seguir, ainda dentro do processo de criação de objeto, inicia-se a execução da função construtora. Se o primeiro comando da construtora for da forma `super(...)`, faz-se a chamada da construtora indicada da superclasse, senão se o primeiro comando for da forma `this(...)`, chama-se a construtora irmã identificada pelos parâmetros dados, e, no caso de o primeiro comando não ser `super` nem `this`, a superclasse deve obrigatoriamente ter uma construtora sem parâmetros que será chamada automaticamente antes executar o corpo da construtora iniciada pelo `new`.

Esse processo se repete para toda construtora chamada dentro da hierarquia, até que a da classe `Object` tenha sido executada. Inicia-se então o processo de retorno, completando a execução das construtoras chamadas. E somente depois do retorno da chamada da construtora da superclasse ou da construtora irmã, é que o corpo da construtora do objeto criado é efetivamente executado.

O exemplo a seguir mostra a ordem em que a execução ocorre, começando pela linha 21 do método `main` da classe `C` declarado na linha 20.

Na linha 22, após a alocação de um objeto do tipo `B`, a construtora declarada na linha 11 é chamada, e o primeiro comando,

o `this(0,1)` da linha 12, transfere o fluxo de execução para a construtora da linha 15.

```
1  class A {
2     public A() {
3        System.out.println("Passei na A()");
4     }
5  }
6  class B extends A {
7     public B (int a) {
8        super();
9        System.out.println("Passei na B(a)");
10    }
11    public B() {
12       this(0,1);
13       System.out.println("Passei na B()");
14    }
15    public B(int a, int b) {
16       System.out.println("Passei na B(a,b)");
17    }
18 }
19 public class C {
20    public static void main (String[]  args) {
21       System.out.println("Comecei na C");
22       B b1 = new B();
23       System.out.println("Voltei na C");
24       B b2 = new B(1,2);
25       System.out.println("Terminei na C");
26    }
27 }
```

Antes de executar o comando da linha 16, a construtora sem parâmetros da classe `A` é automaticamente chamada e, assim, o fluxo é transferido para a linha 2. Nesse momento, a construtora da classe `Object`, superclasse de `A`, é executada, e, após o retorno dessa construtora, imprime-se `Passei na A()`.

Após a conclusão da execução da construtora de `A`, o controle volta à linha 16 para imprimir `Passei na B(a,b)`. E a seguir retorna-se à linha 13, quando imprime-se `Passei na B()`, concluindo a alocação do primeiro objeto `B`.

Para a alocação do segundo objeto `B`, na linha 24, a construtora da linha 15 é chamada, e imediatamente a construtora sem parâmetros da classe `A` é automaticamente acionada e, na sequência, o fluxo é transferido para a linha 2.

Nesse momento, a construtora da classe `Object` é executada e, quando concluída, imprime-se `Passei na A()`. O controle então volta à linha 16 para imprimir `Passei na B(a,b)`, e conclui-se a alocação do segundo objeto `B`.

Resumidamente, a saída do programa é:

```
Comecei na C
Passei na A()
Passei na B(a,b)
Passei na B()
Voltei na C
Passei na A()
Passei na B(a,b)
Terminei na C
```

4.3.1 Iniciação de membros de dados

Como visto, no momento de criação de um objeto de uma dada classe `A`, todos os membros de dados do objeto são iniciados com valores-padrão na ordem de suas declarações.

A seguir as construtoras das classes ancestrais de `A` são executadas na forma e ordem descritas na Seção 4.3.

Somente quando o fluxo de controle retornar da execução dessas construtoras de superclasses, os iniciadores de campos e blocos de inicialização não-estáticos da classe `A` são executados, na ordem

de suas ocorrências, e por último o corpo da construtora de **A** é efetivamente executado.

O programa abaixo ilustra, passo-a-passo, o processo de iniciação dos campos de um objeto do tipo **Pingo**, definido como uma extensão da classe **Ponto**.

```
1  class Ponto {
2     int x = 10, y = 20;
3     Ponto() {x = 1; y = 2;}
4  }
5  class Pingo extends Ponto {
6     int cor = 0xFF00FF;
7  }
```

Como o primeiro comando da construtora de **Ponto** não é uma chamada de construtora, i.e., não é **super(...)** nem **this(...)**, e **Pingo** não tem construtora especificada, o compilador Java transforma as classes **Ponto** e **Pingo**, no momento da geração de código, em texto equivalente a:

```
1  class Ponto {
2     int x = 10, y = 20;
3     Ponto() {
4        super();       // inserido pelo compilador
5        x = 1; y = 2;
6     }
7  }
8
9  class Pingo extends Ponto {
10     int cor = 0xFF00FF;
11     public Pingo() {  // inserido pelo compilador
12        super();       // inserido pelo compilador
13     }                  // inserido pelo compilador
14 }
```

Com a finalidade de acompanhar os passos de iniciação de campos e execução de construtoras, considere o programa `Teste`, definido como:

```
1 public class Teste {
2    public static void main(String[ ] args) {
3       Pingo p = new Pingo();
4    }
5 }
```

O operador `new` da linha 3 de `Teste` cria uma instância de `Pingo`, com espaço para os membros de dados `x`, `y` e `cor`, os quais são imediatamente iniciados com o valor `0`.

A seguir, a construtora `Pingo()` é chamada, e seu primeiro ato é executar a chamada `super()` da linha 12, a qual transfere o controle de execução para a linha 3, dentro de `Ponto`.

O comando da linha 4 de `Ponto` causa a chamada da construtora da classe `Object`, sua superclasse. Quando a execução dessa construtora for concluída, os iniciadores de campo da classe `Ponto` são executados.

Nesse momento, os campos `x` e `y` do objeto `Pingo` que está sendo alocado recebem os valores `10` e `20`, respectivamente.

A seguir, o corpo da construtora de `Ponto` é executado até o seu fim, atribuindo-se `1` e `2`, respectivamente, aos mesmos campos `x` e `y` de objeto `Pingo`.

No retorno à construtora de `Pingo`, na linha 13, os iniciadores de campos da classe `Pingo` são executados, causando a iniciação do campo `cor` com o valor `0xFF00FF`. Com isso, a execução da chamada `Pingo()` é concluída, completando-se a alocação do objeto `Pingo`.

O conhecimento da ordem de execução dos passos de iniciação dos campos de um objeto no momento de sua alocação é muito

importante para se evitar situações de erro e principalmente entender o que ocorre no programa. Um erro possível é o mostrado no programa a seguir, que tenta iniciar um campo com valor ainda não definido.

```
1  class A {
2     private C c;
3     protected A(C c ) {this.c = c;}
4     public void f() { .... c ... }
5  }
6
7  class B extends A {
8     private C y = new C();
9     public B() {super(y);}
10 }
```

O erro é revelado pelo compilador com a mensagem:

`B.java:8: cannot reference y before supertype constructor has been called`,

e deve-se ao fato de o campo `y` da classe `B`, no momento em que é passado para a construtora, no comando `super(y)` da linha 9, ainda não ter sido iniciado com o endereço do objeto `C`, a ser criado no iniciador de campo que está na linha 8.

Isso ocorre porque o processamento de iniciadores de campo somente é feito após o retorno da chamada à construtora da superclasse, e no momento da chamada de `super(y)` na linha 9 da classe `B`, `y` ainda não foi inicializado. Note que a tentativa abaixo de *enganar* o compilador também não funciona.

```
1  class B extends A {
2     private C y;
3     public B() {super(y = new C());}
4  }
```

Nesse exemplo, compilador Java sabiamente protesta com a mensagem:

> `Teste.java:3:cannot reference y before supertype constructor has been called`.

Por outro lado, o seguinte programa compila sem erro:

```
1 class B extends A {
2    private C y;
3    public B() {super(new C());}
4 }
```

4.4 Especialização de classes

Uma importante característica do recurso de hierarquização de classes em linguagens orientadas por objetos é a possibilidade de redefinição da semântica de operações herdadas de forma a tornar o comportamento da subclasse mais especializado do que o da superclasse. O fato de se poder agregar mais atributos na construção da subclasse a torna mais especializada que sua superclasse e isso pode demandar mudanças nas operações herdadas.

Como o comportamento de uma classe é dado por suas operações, redefinir comportamento de uma classe é redefinir o corpo de uma ou mais funções não-finais ou não-estáticas herdadas da superclasse.

A visibilidade, que pode ser `public`, `protected`, `private` ou *pacote*, de um método pode ser aumentada, mas não pode ser reduzida no processo de redefinição.

Consequentemente, nenhum método pode ser redefinido para ser `private`; métodos `public` devem continuar `public` nas classes derivadas, e métodos `protected` somente podem ser redefinidos para `protected` ou `public`, mas nunca para `private`.

Redefinição de um método com tipos ou número de parâmetros diferentes dos do método herdado é permitido, mas, nesse caso, não se trata efetivamente de redefinição de métodos da superclasse, mas da introdução de um novo método homônimo, sobrecarregando um nome antigo. O compilador usa os tipos dos parâmetros para diferenciá-los.

As redefinições efetivas de um dado método devem ter exatamente a mesma assinatura do método herdado, exceto que o seu tipo de retorno pode ser especializado. Essas duas possibilidades são conhecidas como **regra da invariância** e **regra da semi-invariância**, respectivamente.

No exemplo abaixo, as funções `f` e `g` da classe `D` foram redefinidas na subclasse `E`.

A redefinição de função `f` obedece a regra da invariância, sendo preservados todos os tipos na assinatura do `f` declarado em `D`.

Por outro lado, a redefinição de `g` segue a regra da semi-invariância, i.e., invariância para os parâmetros e usa co-variância para o tipo do valor de retorno.

```
1  class A { ... }
2  class B extends A { ... }
3  class C1 { ... }
4  class C2 { ... }
5  ...
6  class Cn { ... }
7  class D {
8     A f(C1 x1, C2 x2, ..., Cn xn) { ... }
9     A g(C1 y1, C2 y2, ..., Ck yk) { ... }
10 }
11 class E extends D {
12    A f(C1 z1, C2 z2, ..., Cn zn) { ... }
13    B g(C1 t1, C2 t2, ..., Ck tk) { ... }
14 }
```

O programa `RedefineFunções` abaixo mostra a flexibilidade e as restrições oferecidas pelo mecanismo de redefinição.

```
1  public class RedefineFunções {
2     public static void main(String[] args) {
3        D x = new E(); ...
4        E z = new E(); ...
5        A y = x.f();  // OK
6        A y = x.g();  // OK
7        B w = x.g();  // Erro de compilação
8        B v = z.g();  // OK
9        ...
10    }
11 }
```

Note que embora a redefinição de `g` na classe `E` retorne um objeto do tipo `B`, o valor retornado por `x.g()` nas linhas 6 e 7 de `RedefineFunções` é sempre suposto ser do tipo `A`, pois o tipo estático de `x` é `D`, e o compilador não pode garantir que o tipo dinâmico de `x` seja `E`.

Adota-se sempre a solução mais conservadora, que é usar o tipo estático de uma referência para determinar as operações aplicáveis ao objeto apontado. Por essa regra, supõe-se que `z.g()` retorne um objeto do tipo `B` para fins de determinar a validade da operação.

Especialização de classes pode produzir mais de uma definição para uma mesma operação. Dessa forma, podem conviver em um mesmo programa mais de uma versão de uma mesma operação. Isto cria a necessidade de se determinar a cada ativação de uma operação que versão do método que a implementa deve ser escolhida para a execução. Para isto, são usados os conceitos de tipo estático e tipo dinâmico de uma referência.

O tipo estático da referência de um objeto receptor provê as informações necessárias para verificar a validade do envio de mensagens ao objeto. O compilador usa essas informações para validar

a qualificação de membros do objeto a partir de uma referência e identificar o tipo do membro referenciado.

O tipo dinâmico da referência é usado em tempo de execução para determinar que método deve ser executado em resposta a uma mensagem enviada ao objeto. O mecanismo usado para resolver esse problema é chamado ligação dinâmica de operações.

4.4.1 Implementação da ligação dinâmica

Para ilustrar o funcionamento do mecanismo de ligação dinâmica, considere a hierarquia definida pelas classes `A`, `B` e `C` apresentadas ao longo desta seção. Suponha que a classe `A` tenha o formato:

```
1 class A {
2    private int a;
3    public void f (int x) {...} // void Fa(A this,int x)
4    public void g (int y) {...} // void Ga(A this,int y)
5    private void p(int z) {...} // void Pa(A this,int z)
6 }
```

Suponha que os métodos `f`, `g` e `p` da classe `A` sejam traduzidos para as funções `Fa`, `Ga` e `Pa`, codificadas em *bytecode* por funções cujas assinaturas, na linguagem **C** [28], correspondem às mostradas nos comentários das linhas de 3 a 5. Note que as funções traduzidas para **C** têm um parâmetro adicional, de nome `this`, usado para receber a referência do objeto qualificador da chamada da função.

Para cada classe usada no programa, independentemente do número de vezes, carrega-se, durante a execução, seu descritor na forma de um objeto do tipo `Class`, que contém, entre outros elementos, informação sobre todos os membros da classe e uma tabela denominada Tabela de Métodos Virtuais (**vmt**). Para a classe `A` definida acima, o objeto `Class` alocado tem o formato mostrado

na Fig. 4.11, onde a tabela **vmt** contém entradas para `f` e `g`, instaladas logo abaixo do campo que identifica o tipo do objeto.

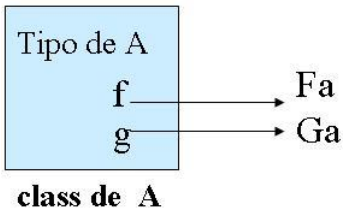


Figura 4.11 Descritor da Classe A

Na tabela **vmt** são alocados os endereços da primeira instrução de cada método redefinível da classe. Os métodos redefiníveis são numerados pelo compilador Java a partir de 1 (um), e o número de cada um é usado para indexá-lo na tabela **vmt**. Essa numeração deve ser mantida consistente em toda a hierarquia da classe. No exemplo, `f` é associado a `1`, e `g`, a `2`. Métodos privados ou estáticos não podem ser redefinidos, e assim nunca são incluídos na **vmt** da classe. Em geral, o código *bytecode* desses métodos são alocados junto ao descritor da classe, em local conhecido pelo compilador.

O programa `M1` a seguir aloca um objeto do tipo `A` e atribui seu endereço à referência `r`. Todo objeto possui implicitamente um tipo dinâmico, que é o endereço, referido aqui como `vmtr`, do descritor de sua classe, conforme mostra o leiaute da Fig. 4.12.

```
1 public class M1 {
2    public static void main(String[ ] args) {
3       A r = new A();
4       r.f(5);    // ( *(r->vmtr[1]) )(r,5);
5       r.g(5);    // ( *(r->vmtr[2]) )(r,5);
6    }
7 }
```

Os comandos das linhas 4 e 5 de `M1` são traduzidos para instruções em *bytecode* com semântica equivalente aos códigos, na

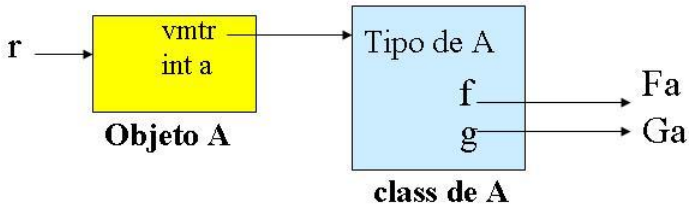


Figura 4.12 Leiaute dos Objetos de `M1`

linguagem **C** [28], mostrados nos respectivos comentários nessas linhas, das quais depreende-se que a execução do comando `r.f(5)` segue os seguintes passos:

1. usa-se a referência `r` para ter acesso ao objeto alocado
2. o campo `vmtr` do objeto alocado fornece o endereço do descritor da classe do objeto, que tem a tabela **vmt** logo após o campo de tipo
3. o número de ordem de `f`, no caso `1`, é usado para indexar o vetor `vmt` do descritor da classe e obter o endereço `vmtr[f]` da primeira instrução da função `Fa` a ser executada
4. a seguir, efetua-se a chamada `vmtr[f](r,5)`, a qual é passado como primeiro parâmetro o endereço do objeto corrente.

A tabela **vmt** de uma subclasse é construída a partir da tabela correspondente herdada da superclasse, onde definem-se novos valores para as entradas correspondentes aos métodos redefinidos e, possivelmente, acrescentam-se novas entradas referentes à declaração de novos métodos.

Considere agora a classe `B`, que estende `A` e redefine o método `g`.

```
1 class B extends A {
2    private int b;
3    public void g (int y) {...} // void Gb(B this,int y)
4    private void p(int z) {...} // void Pb(A this,int z)
5 }
```

A declaração de método privado `p` na classse `B` apenas cria um novo método totalmente desvinculado do método homônimo em `A`. Por outro lado, a redefinição de `g` em `B` é compilada para `Gb`, e no descritor da classe `B`, a entrada `g` da respectiva tabela **vmt** refere-se a essa nova definição.

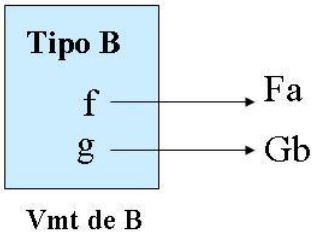


Figura 4.13 Descritor da Classe B

O leiaute de objetos do tipo `B` é exibido na Fig. 4.13, onde pode-se ver que a entrada correspondente ao método redefinido `g` tem o endereço do código de `Gb`, enquanto a entrada `f` é uma cópia de sua correspondente na Fig. 4.11.

O programa `M2` a seguir aloca um objeto do tipo `B` e cria para ele duas referências, `r` e `s`, de tipos estáticos `A` e `B`, respectivamente, conforme mostra a Fig. 4.14. Essas duas referências apontam para o mesmo objeto, e, portanto, têm o mesmo tipo dinâmico.

Consequentemente, os comandos `s.g(5)` e `r.g(5)` das linhas 5 e 6 causam a ativação da mesma versão compilada de `g`, a identificada por `Gb` na Fig. 4.14.

```
1 public class M2 {
2    public static void main(String[ ] args) {
3       B s = new B();   A r = s;
4       s.f(5);    // ( *(s->vmtr[1]) )(s,5);
5       s.g(5);    // ( *(s->vmtr[2]) )(s,5);
6       r.g(5);    // ( *(r->vmtr[2]) )(r,5);
7    }
8 }
```

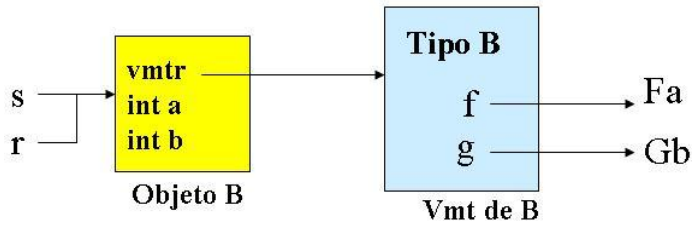


Figura 4.14 Leiaute dos Objetos de `M2`

Para completar o quadro, considere a classe `C`:

```
1 class C extends B {
2    private int c;
3    public void h (int z) {...} // void Hc(C this,int z)
4 }
```

O descritor de `C`, mostrado na Fig. 4.15, herda a **vmt** do descritor de `B`, e a ela acrescenta uma nova entrada para o método `h` declarado em `C`. Essa entrada é iniciada como o endereço de primeira instrução de `Hc`, que é o código *bytecode* de `h`.

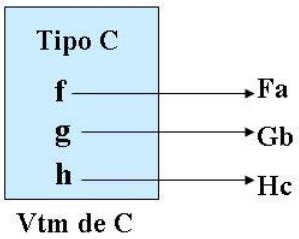


Figura 4.15 Descritor da Classe C

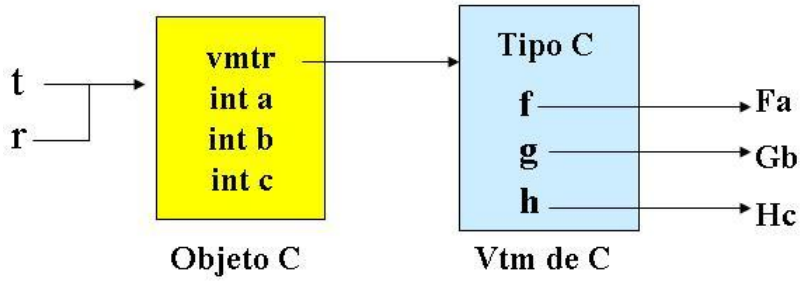


Figura 4.16 Leiaute dos Objetos de `M3`

O programa `M3` abaixo cria uma instância de um objeto do tipo

`C` e guarda seu endereço nas referências `t` e `r` de tipos estáticos `C` e `A`, respectivamente.

Qualquer chamada de método qualificado pelas referências `t` ou `r`, com a configuração descrita na Fig. 4.16, ativa a respectiva função compilada identificada pela **vmt** do descritor de `C`.

```
1 public class M3 {
2    public static void main(String[ ] args) {
3       C t = new C();  A r = t;
4       t.f(5);   // ( *(t->vmtr[1]) )(t,5);
5       t.g(5);   // ( *(t->vmtr[2]) )(t,5);
6       r.g(5);   // ( *(r->vmtr[2]) )(r,5);
7       t.h(5);   // ( *(t->vmtr[3]) )(t,5);
8    }
9 }
```

Em resumo, há um objeto descritor alocado automaticamente para cada classe que aparecer no programa. Todos os objetos de uma dada classe apontam para um mesmo descritor. Esse apontador identifica univocamente o tipo dinâmico de cada objeto. Os métodos redefiníveis de cada classe têm o endereço de seu código compilado instalado na tabela `vmt` do objeto descritor da classe. O índice de cada método na `vmt` é preservado ao longo da hierarquia de classes, i.e., se um método da classe `A`, definido pela primeira vez na hierarquia, ocupa a posição `k` na `vmt` de `A`, então ele ocupará a mesma posição `k` na `vmt` de todas as classes descendentes de `A`.

4.4.2 Redefinição de métodos finais

Métodos declarados finais têm uma restrição importante: eles não podem ser redefinidos nem redeclarados nas classes estendidas. Esse recurso é usado quando se quer assegurar que a semântica de uma operação do tipo não possa ser alterada.

As tentativas que vão de encontro a essa regra serão apontadas como erro pelo compilador, como no programa a seguir, onde a definição de `f` na classe `B` é rejeitada.

```
1 public class A {
2     private int x;
3     final public void f(int x) {this.x = x;}
4 }
5 public class B extends A {
6     private int x;
7     public void f(int x) {this.x = x*x;} // <== Erro
8 }
```

Note que um método declarado final não pode mais ser redefinidos dentro da hierarquia. Um método final é normalmente inserido na **vmt** de sua classe, mas o valor de sua entrada nessa tabela não pode ser alterado quando subclasses forem criadas.

4.4.3 Redefinição de métodos estáticos

Métodos estáticos não são redefiníveis, mas pode-se declarar nas subclasses outros métodos com mesmo nome. Nesse caso, essas redeclarações são criações de funções homônimas independentes.

Essa prática pode levar a conflito de nomes entre a versão antiga e a versão nova do elemento redeclarado. Nesses casos, a ambiguidade de acesso a nomes de membros estáticos que colidem deve ser resolvida via sua qualificação pelo nome de suas respectivas classes.

4.4.4 Redefinição de membros de dados

Membros de dados não podem ser redefinidos. A redeclaração de um atributo da superclasse na subclasse que a estende é permitido, mas apenas tem o efeito de criar um novo atributo na subclasse, a qual passa a ter dois atributos com mesmo nome. E o nome

local oculta o nome herdado em referências sem qualificação. O exemplo a seguir ilustra os procedimentos adotados para identificar os membros de dados desejados.

```
1 class A {
2    public String s = "A";
3    public void h() { System.out.print(" A: " + s);}
4 }
5 class B extends A {
6    public String s = "B";
7    public void h() { System.out.print(" B: " + s);}
8 }
```

Nesse exemplo, se o método `h` da classe `A` for executado, o campo `s`, que é o mesmo que `this.s`, referenciado na linha 3 é o declarado na linha 2. Por outro lado, se o método em execução for o `h` da linha 7, o campo `s` nele referenciando é o da linha 6. Em outros casos, os conflitos de nomes são resolvidos pelo tipo estático. No caso de referências explicitamente qualificadas, como as das linhas 6 e 7 da classe `OcultaDados` abaixo, usa-se o tipo estático da referência para identificar o campo referenciado.

```
1 public class OcultaDados {
2    public static void main(String args) {
3       B b = new B();  A a = b;
4       a.h();
5       b.h();
6       System.out.print(" a.s = " + a.s);
7       System.out.print(" b.s = " + b.s);
8    }
9 }
```

O programa `OcultaDados`, que gera a alocação de objetos detalhada na Fig. 4.17, produz a saída: `B: B B: B a.s = A b.s = B`

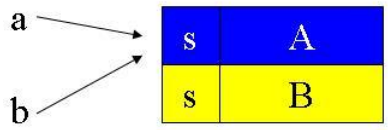


Figura 4.17 Leiaute dos Objetos de `OcultaDados`

4.4.5 Modificadores de acesso

As nuances do funcionamento dos mecanismos de redefinição, ligação dinâmica e tipos dinâmicos de Java podem ser vistas no esquema de programa abaixo, onde pretende-se considerar os casos de o modificador, representado por `<modificador>`, associado ao método `increment`, ser `private`, *pacote*, `protected`, `public` ou `static`.

```
1  publi class A {
2     public int i = 10;
3     <modificador> void increment() {i++;}
4     public void display() {
5        increment();
6        System.out.println("i = " +  i);
7     }
8  }
9  public class B extends A {
10    public int i = 20;
11    public void increment() {i++;}
12 }
13 public class H {
14    static public void main(String[ ] args) {
15       B x = new B();
16       x.display();
17    }
18 }
```

O fluxo de execução do programa `H` para cada especificação de `<modificador>` do método `increment` declarado na linha 3 é o seguinte:

- `private void increment()`:
  Nesse caso, o método da linha 11 não é uma redefinição do método homônimo da linha 3. O fluxo de execução do programa é o seguinte: 14, 15, 16, 4, 5 e 3. Nesse momento, o valor `A.i`, inicialmente `10`, é incrementado e o controle volta à linha 6, que imprime a resposta `i = 11`.
- `void increment()`:
  Se as classes `A` e `B` estiverem em pacotes distintos, a visibilidade *pacote* de `increment` é equivalente a `private`, e, portanto, o programa comporta-se como no item anterior. Se, entretanto, essas duas classes pertencerem ao mesmo pacote, o método da linha 11 é uma redefinição do método homônimo da linha 3. Nesse caso, o fluxo de execução do programa é o seguinte: 14, 15, 16, 4, 5 e 11. Nesse momento, valor de `B.i`, inicialmente `20`, é incrementado, e o controle retorna à linha 6, que imprime a resposta `i = 10`.
- `protected` ou `public void increment()`:
  O método da linha 11 é uma redefinição do método homônimo da linha 3, independentemente dos pacotes em que `A` e `B` estejam. O fluxo de execução do programa é o seguinte: 14, 15, 16, 4, 5 e 11. Nesse momento, valor de `B.i`, inicialmente `20`, é incrementado, e o controle volta à linha 6, que imprime a resposta `i = 10`.
- `static void increment()`:
  Nesse caso, o método da linha 11 não é uma redefinição do método homônimo da linha 3. O fluxo de execução do programa é o seguinte: 14, 15, 16, 4, 5 e 3. Nesse momento, o valor de `A.i`, inicialmente `10`, é incrementado, e a execução continua na linha 6, que imprime a resposta `i = 11`.

4.5 Hierarquia múltipla

Em Java, toda classe estende direta ou indiretamente `Object`, mas interfaces não têm raiz comum pré-definida. Toda classe sempre estende uma e somente uma superclasse, mas pode implementar simultaneamente várias interfaces. Portanto, tem-se apenas herança simples de classes, e herança múltipla surge quando a classe implementa uma ou mais interfaces. As interfaces implementadas e a classe que for estendida são consideradas supertipos da classe.

O formato geral da declaração de uma classe é:

```
[<modificadores>] class Nome [extends Superclasse]
        [implements Interface1, Interface2, ...] {
     declarações de métodos, campos e iniciadores
}
```

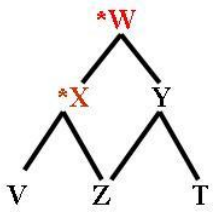


Figura 4.18 Hierarquia de Classes e Interfaces

O mecanismo de estender classes e implementar interfaces geram uma hierarquia de tipos, conforme exemplifica a Fig. 4.18, onde interfaces estão marcadas com `*`. E essa hierarquia corresponde ao seguinte esquema de declarações:

```
1 interface W { ... }
2 interface X extends W { ... }
3 class Y implements  W { ... }
4 class Z implements X extends Y { ...  }
5 class V implements X { ... }
6 class T extends Y { ... }
```

Observe que a classe `Z` tem relacionamento **é-um** com a superclasse `Y` e com a interface `X`, permitindo a objetos do tipo `Z`

comportar-se como objetos do tipo `Y` ou de acordo com a interface `X`. Por isso, a classe `Z` deve implementar os métodos de `X` e reusar a implementação de `Y`. Métodos de `Y` cujas assinaturas coincidem com alguma dentre as declaradas em `X` automaticamente implementam os correspondentes na interface, a menos que sejam redefinidos em `Z`.

Protótipos de métodos homônimos, mas com diferentes assinaturas, especificados em várias interfaces, e implementados por uma mesma classe, são tratados como casos de sobrecarga de nomes. Se, entretanto, eles tiverem a mesma assinatura, incluindo o mesmo tipo de valor de retorno, trata-se de dois protótipos da mesma função. Um único método implementa todos protótipos com mesma assinatura.

Conflito somente ocorre quando mais de um protótipo têm a mesma assinatura, mas diferentes tipos de valor de retorno. Nesse caso, considera-se que as duas interfaces não são implementáveis juntas, porque não há como escrever um método que atenda a ambas interfaces. A escrita de mais de um método que se diferencia um do outro apenas no tipo do valor de retorno não é permitida em um mesmo escopo em Java.

Se, entretanto, protótipos de métodos com mesmo nome diferem-se apenas nos tipos das exceções (vide Capítulo 7) que suas implementações podem lançar, então uma implementação comum somente pode lançar as exceções que sejam comuns a todos os protótipos. Colisões de nomes de constantes, i.e., membros estáticos finais, são permitidas e devem ser resolvidas via qualificação explícita das constantes com os nomes das respectivas interfaces.

Um exemplo concreto de hierarquia múltipla em Java é a modelagem da situação, definida a seguir pelas classes `Campo`, `Data` e `CampoDeData`, onde permite-se realizar sobre os objetos do tipo

`Campo`, as operações `edita` e `limpe`, e sobre os objetos do tipo `Data`, as operações `avance` e `defina`.

```
1  public class Campo {
2      ...
3      public Campo(...) { ...  }
4      public void edita(...) {...}
5      public void limpe(...) {...}
6  }
7  public class Data implements DataI {
8      ...
9      public void defina(...) {...}
10     public void avance(...) {...}
11 }
12 public interface DataI {
13      public void defina(...);
14      public void avance(...);
15 }
16 public class CampoDeData extends Campo implements DataI {
17       ...
18       Data d;
19       public CampoDeData(Data d) {this.d = d;}
20       public void avance(...) {d.avance();}
21       public void defina(...) {d.defina();}
22       public void mudeCor(...) {...}
23 }
```

A classe `CampoDeData` têm comportamento dual, pois trata-se de um campo de registro, herdeiro de `Campo`, e que também é uma data, pois implementa `DataI`. Sobre ele aplicam-se igualmente as operações de `Campo` e as de `Data`.

Note que as operações de `Campo` são herdadas, mas as de `DataI` precisam ser implementadas, aproveitando-se das operações da `Data`.

4.6 Conclusão

A herança é um mecanismo para reúso de implementação e construção de hierarquia de tipos. Réuso ocorre quando novas classes são criadas a partir de classes existentes.

Em Java, permite-se apenas herança simples de classes. Herança múltipla em Java está restrita ao conceito de interface de classe, que define tipos de dados abstratamente, sem qualquer informação sobre sua implementação.

É fato que herança simples atende à maior parte das aplicações, e a proibição de herança múltipla na linguagem facilita a implementação de seu compilador. Entretanto, a ausência de herança múltipla de classes em Java limita o poder de expressão da linguagem, no sentido em que há situações em que o uso de herança múltipla na modelagem seria o mais natural. Por outro lado, o uso de interface permite reduzir parcialmente essa perda.

Algumas linguagens, como C++ [53] e Eiffel [40], permitem herança múltipla de classes, mostrando que sua implementação é factível, mas os mecanismos subjacentes necessários são complexos.

Para compreender essa complexidade introduzida na implementação do compilador pela herança múltipla de classes, suponha, por um instante, que esse recurso fosse permitido em Java, ou seja, suponha que a declaração `class C extends A, B {...}` da Fig. 4.19 seja válida.

Suponha também que o código da linha 6 a 11 da Fig. 4.19 ocorra em algum método usuário das classes `A`, `B`, `C` e `D`, que, ao ser chamado, executa a linha 7, gerando a alocação de objetos indicada na mesma figura.

A primeira dificuldade está na resolução do conflito de acesso ao campo `x`, de objetos do tipo `C`, que, por herança, possuem dois

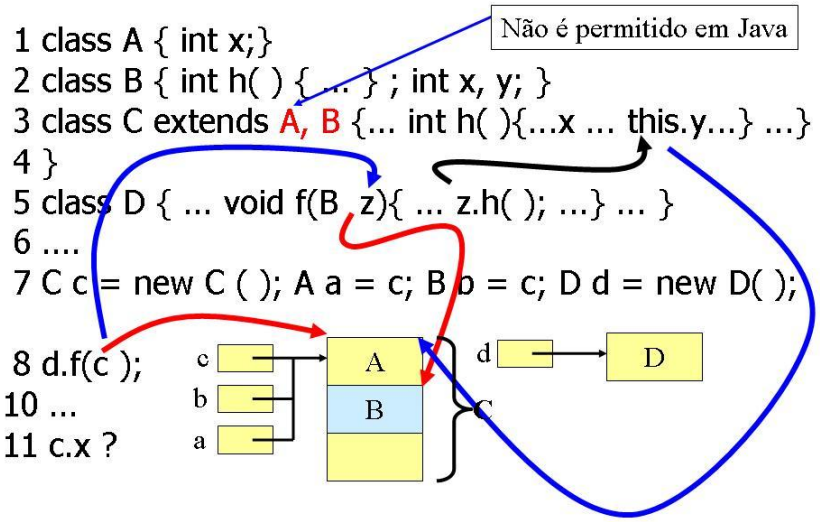


Figura 4.19 Objeto Corrente com Herança Múltipla

campos com esse nome. Não há como distinguir um do outro sem alterar a linguagem Java para permitir algum tipo de qualificação mais explícita, na qual se indique a origem do campo, isto é, se é o que veio de `A` ou o de `B`.

O segundo problema tem a ver com o valor da referência implícita `this`. Ressalte-se que o objeto apontado por `c` tem três partes: a inicial, herdada de `A`, a do meio, herdada de `B`, e os campos adicionais, acrescentados pela declaração de `C`. Naturalmente, a referência `c` aponta para o primeiro *byte* do objeto referenciado.

No comando `d.f(c)` da linha 8, o valor da referência `c` é passado para o parâmetro formal `z` do método `D.f`. Entretanto o valor de `c` não pode ser simplesmente copiado, porque o parâmetro formal é do tipo `B`, e o objeto do tipo `B` referenciado por `c` tem endereço $\texttt{c} + \Delta$, onde Δ é o tamanho da parte `A` de objeto apontado por `c`. Portanto, imediatamente após a entrada no método `D.f`, tem-se $\texttt{z} = \texttt{c} + \Delta$.

A seguir, o comando `z.h()` causa a transmissão do valor da referência `z` para o parâmetro implícito `this`, do tipo `C`, do método `C.h`. O valor passado não pode ser o endereço da parte `B`, mas a

do objeto integral apontado, i.e., o valor passado deve ser corrigido para `z` - Δ.

Esses detalhes técnicos dificultariam muito a implementação do compilador, justificando a decisão de não ter em Java herança múltipla de classes.

Exercícios

1. Por que métodos estáticos ou finais e também métodos com visibilidade `private` não podem ser declarados abstratos?
2. Como seria a tabela de métodos virtuais se fosse permitida herança múltipla de classes em Java?
3. Qual seria a perda de poder de expressão se Java não tivesse o mecanismo de interfaces?
4. Qual a diferença entre uma interface e uma classe abstrata sem qualquer campo e somente com os mesmos protótipos da interface?

Notas bibliográficas

O conceito de hierarquia de classes é bem discutido em praticamente todos os textos sobre Java, e em particular no livro de K. Arnold, J. Gosling e D. Holmes [2, 3].

A técnica de implementação de tabelas de métodos virtuais e de herança múltipla aqui descrita é a utilizada no compilador da linguagem C++. Seus detalhes são descritos por M. Ellis e B. Stroustrup [16]. Java usa a mesma solução, exceto por detalhes de implementação.

Capítulo 5

Classe `Object`

Toda classe em Java pertence à hierarquia cuja raiz é a classe `Object`, a qual provê um conjunto de operações que, consequentemente, pertencem a todo e qualquer objeto. Toda classe que não estende explicitamente outra estende implicitamente a classe `Object`.

5.1 Operações de `Object`

As operações definidas pela classe `Object` são as seguintes:

- `public boolean equals(Object obj)`:
  compara bit-a-bit objetos receptor e parâmetro `obj`.
- `public int hashCode( )`:
  retorna o código hash do objeto receptor. Objetos distintos de mesma classe geram *hashcode* distintos.
- `protected` Object clone( ):
  cria uma cópia ou clone do objeto receptor e retorna a referência ao objeto criado Trata-se de uma cópia *rasa*[1].
- `public final Class getClass( )`:
  retorna o objeto que descreve a classe do objeto receptor. A classe `Class` é discutida em mais detalhes no Capítulo 16.

[1]Cópia rasa copia bit-a-bit todos os bits da representação de um objeto, e não segue as referências encontradas.

- `protected void finalize( )`:
  finaliza o objeto receptor imediatamente antes de ele ser recolhido pelo coletor de lixo (operação depreciada).
- `public String toString( )`:
  retorna uma cadeia de caracteres que representa o objeto receptor. A versão `toString` definida em `Object` retorna uma cadeia com o símbolo `@` seguido do *hash code* do receptor.
- `final void wait(long timeout) throws InterruptedException`:
  a *thread* que emitir a operação espera ser notificada, via o objeto corrente, mas espera no máximo o tempo especificado `timeout` (ms). Se `timeout==0` espera indefinidamente.
- `final void wait() throws InterruptedException`:
  o mesmo que `wait(0)`.
- `final void wait(long mili, int nanos) throws InterruptedException`:
  o tempo de espera é `mili*1000 + nanos`.
- `public final void notify()`:
  notifica uma das *threads* que esperam por uma condição na fila de espera do objeto monitor corrente do `notify`. Não é possível escolher quem será notificado. Use com cuidado.
- `public final void notifyAll()`:
  notifica todas as *threads* que esperam por alguma condição.

As operações acima são herdadas por todas as classes e portanto são aplicáveis a qualquer objeto criado em um programa. Os métodos `wait`, `notify` e `notifyAll`, usados em sincronização de *threads* estão detalhados no Capítulo 17.

5.2 Clonagem de objetos

Uma atribuição, como a da linha 3 no trecho de programa abaixo, apenas copia o valor armazenado no endereço de memória dado pela expressão do lado direito do comando de atribuição, no caso o endereço denotado pela variável `a`, para a posição de memória denotada pelo lado esquerdo, no caso a referência `b`.

```
1 class A {int x = 10; int y = 20;}
2 A a = new A( );
3 A b = a;
4 ...
```

Trata-se portanto de uma atribuição de uma referência e não do objeto referenciado. Assim, o efeito da execução da atribuição da linha 3 é ilustrado pela Fig. 5.1. Há, contudo, situações que se deseja o efeito mostrado pela Fig. 5.2.

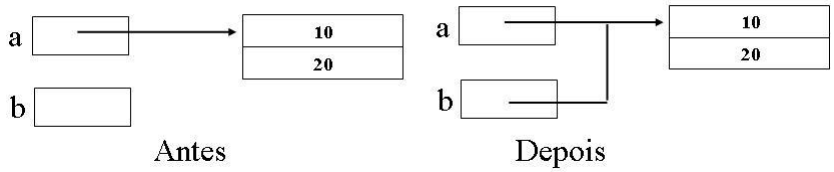


Figura 5.1 Atribuição de referência

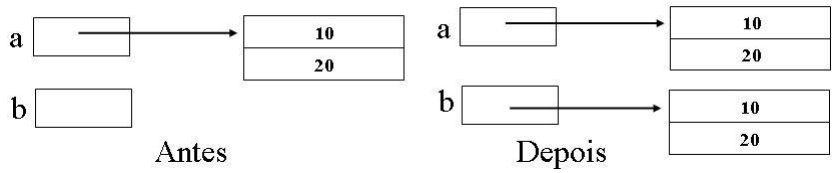


Figura 5.2 Cópia de objetos

Note que, nesse caso, uma cópia ou clone do objeto referenciado por `a` é efetivamente criado. Para obter esse efeito, a classe `A` deve ser redefinida apropriadamente para aceitar clonagem de seus objetos e, no lugar da atribuição `b = a`, deve-se usar `b = a.clone()`.

Toda classe possui, por herança de **Object**, o método **clone**, que faz duplicação bit-a-bit do objeto receptor, isto é, **a.clone()** faz uma cópia bit-a-bit do objeto apontado por **a** e devolve o endereço da cópia.

Considere a classe **Dia**, definida como **Cloneable** para que **clone** herdado possa ser redefinido:

```
1  public class Dia implements Cloneable {
2     private int, dia, mes, ano;
3     public Dia(int dia, int mes, int ano) {
4         this.dia = dia; this.mes = mes; this.ano = ano;
5     }
6     public void altera(int dia, int mes, int ano) {
7         this.dia = dia; this.mes = mes; this.ano = ano;
8     }
9     public Object clone() {return super.clone( );}
10 }
```

Nesse exemplo, a redefinição de **clone**, herdado de **Object**, apenas amplia a visibilidade desse método para que ele possa ser chamado por usuários da classe **Dia**. A cópia é feita pelo **clone** herdado de **Object**. O programa **TestaDia** mostra o funcionamento da clonagem:

```
1  public class TestaDia {
2    public static void main(String[ ] args) {
3        Dia d1 = new Dia(30,3,2004);
4        Dia d2 = d1;
5        Dia d3 = (Dia) d1.clone( );
6        d1.altera(1,4,2004);
7    }
8  }
```

Note, na Fig. 5.3, que os objetos apontados por **d1** e **d2** são compartilhados, mas **d3** referencia um objeto distinto.

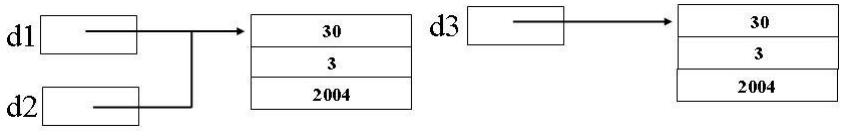


Figura 5.3 Clonagem de Dia (cópia rasa)

Como a cópia é *rasa* (**shallow copy**), durante a cópia ponteiros não são seguidos, a clonagem gera compartilhamento dos objetos apontados por campos do objeto copiado. Caso esse não seja o efeito desejado, o programador deve redefinir o método `clone` herdado para propagar a operação de cópia aos objetos alcançáveis a partir do objeto copiado. Essa redefinição somente é permitida em classes que implementam a interface `Cloneable`. A violação dessa regra causa o levantamento de uma exceção.

Considere agora a classe `Empregado` definida abaixo como clonável, e que é usada para mostrar como propagar a operação de cópia aos objetos alcançáveis a partir do objeto copiado. Inicialmente, a clonagem será rasa e depois será modificada adequadamente para seguir os ponteiros.

```
1  public class Empregado implements Cloneable {
2     private Dia dataContratacao;
3     private String nome;
4     public Empregado(String nome, Dia data) {
5        this.nome = nome;
6        this.dataContratacao = data;
7     }
8     public String getname() {return nome;}
9     public Object clone( ) {
10       Empregado e = (Empregado) super.clone( );
11       return e;
12    }
13 }
```

A execução do programa `TestaEmpregado` gera a Fig. 5.4.

```
1 public class TestaEmpregado {
2   public static void main(String[ ] args) {
3       Dia data = new Dia(11,8,2008);
4       Empregado e1 = new Empregado("Maria", data);
5       Empregado e2 = e1;
6       Empregado e3 = (Empregado) e1.clone( );
7   }
8 }
```

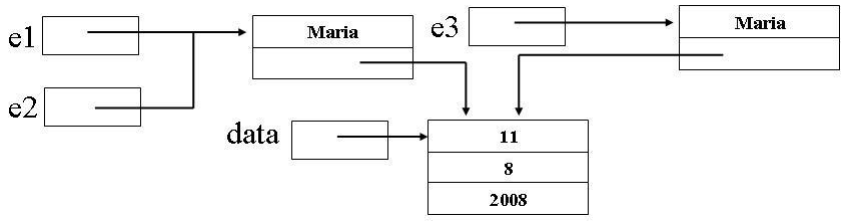


Figura 5.4 Clonagem de Empregado (cópia rasa)

Para evitar o compartilhamento da `dataContratação` pelos objetos clonados, deve-se reescrever o método `clone`, para além da clonagem do objeto corrente, seguir a referência `dataContratação` para a clonagem do objeto associado.

```
1 public class Empregado implements Cloneable {
2    private Dia dataContratacao;
3    private String nome;
4    public Empregado(String nome, Dia data) {
5      this.nome = nome; this.dataContratacao = data;
6    }
7    public String getname( ) {return nome;}
8    public Object clone( ) {
9       Empregado e = (Empregado) super.clone( );
10      e.dataContratacao = (Dia)dataContratacao.clone( );
11      return e;
12   }
13 }
```

Após a execução do mesmo programa `TestaEmpregado`, definido acima, o leiaute dos objetos é o mostrado na Fig. 5.5.

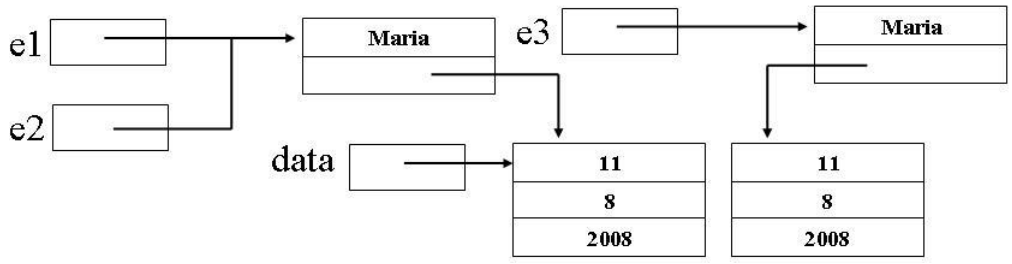


Figura 5.5 Clonagem de Empregado (cópia profunda)

5.3 Conclusão

A classe `Object` é a mãe de todas as classes e disponibiliza um conjunto de operações muitos úteis para, por exemplo, administrar sincronização de *threads*, fazer clonagem de objetos e apoiar refexão computacional. Um pouco de tudo isso será visto nos próximos capítulos.

Exercícios

1. Qual é a utilidade de toda classe herdar as operações de `Object`?

Notas bibliográficas

A melhor referência bibliográfica para conhecer em profundidade a estrutura da classe `Object` em Java é a página oficial da Oracle para a linguagem Java (http://www.oracle.com).

O livro de Ken Arnold et alii [3] é também uma referência importante e de fácil leitura.

Capítulo 6

Polimorfismo

Polimorfismo é um conceito associado à ideia de entidades assumirem mais de uma forma ou que tenham mais de uma semântica associada. Isso ocorre quando uma referência designa um objeto de tipo distinto do que foi declarada, uma função funciona bem com parâmetro de diferentes tipos ou quando, no mesmo contexto, um mesmo nome é associado a diferentes funções.

Essencialmente, polimorfismo permite que nomes sejam usados via suas interfaces independentemente de suas implementações. Pode-se então afirmar que polimorfismo é a habilidade de se esconder diferentes implementações atrás de interfaces comuns. Essa habilidade permite aumentar o grau de reúso de código e a legibilidade de programas.

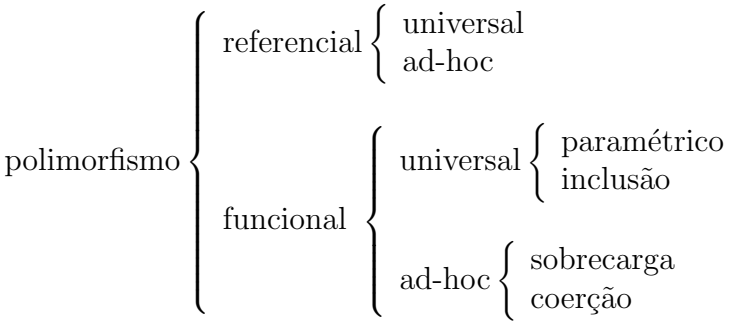


Figura 6.1 Classificação de Polimorfismo

O termo polimorfismo, no contexto de linguagens de programa-

ção, foi introduzido e classificado por Cristhropher Strachey em 1967 [51, 52]. Em 1985, Luca Cardelli e Peter Wegner[4] modificaram a classificação de Strachey, introduzindo novas categorias. Atualmente, o polimorfismo encontrado nas linguagens de programacão modernas orientadas por objetos pode ser classificado conforme a Fig. 6.1.

6.1 Polimorfismo referencial

Linguagens de programação imperativas tradicionais permitem que apontadores contenham endereços de dados de diferentes tipos. Apontadores com essa flexibilidade são ditos polimórficos. Se puderem apontar para qualquer tipo, são polimórficos universais. Se estiverem limitados a tipos específicos e predefinidos, são considerados polimórficos ad-hoc.

Por exemplo, um apontador do tipo `void*` de C++ [53] pode receber valores de apontadores de diversos tipos, mas não de todos os tipos, pois não podem receber endereços de funções ou de membros de classe, sendo assim um exemplo de referência polimórfica ad-hoc. Além disso, o uso de apontadores `void *` somente tem validade em casos específicos e demanda controle de uso e conversão explícita de dados por parte do programador.

Referências em Java são recursos de nível mais elevado do que as da linguagem C[28]. São como os apontadores de linguagens imperativas tradicionais, como C, mas sem a sintaxe e a flexibilidade usuais dos apontadores.

Em Java, o polimorfismo de referência surge do uso do modelo Semântica de Referência, o qual trata da manipulação e da composição de objetos de um programa baseado no uso de referências para esses objetos, em vez de seus valores.

Esse modelo se contrapõe à chamada semântica de valor, na qual usar-se-iam os valores dos objetos em operações como atribuição, passagem de parâmetro e composição de objetos.

Linguagens orientadas por objetos em geral dependem dos recursos providos por apontadores para expressar de uma forma eficiente muitas operações, como composição de objetos e compartilhamento de dados.

Note que sempre que mais de um objeto tiverem partes em comum é mais fácil e eficiente manter referências para essas partes do que copiá-las em cada objeto. Além disso, o uso de referências no lugar de cópias de valores facilita a manutenção da coerência semântica dos dados, porque, quando uma parte comum a vários objetos sofrer alguma modificação, não é necessário percorrer todos os objetos onde a informação compartilhada estiver associada para se fazer as devidas alterações.

Java, por razões pragmáticas relacionadas à eficiência de armazenamento de dados, adota semântica de referência somente para objetos. Valores de tipos primitivos são armazenados e operados usando semântica de valor.

Assim, objetos em Java podem ser manipulados apenas via suas referências e, uma vez criados, nunca mudam de forma ou tipo. Todo objeto tem tamanho e leiaute predefinidos no momento de sua alocação, e esses atributos não mais são alterados durante a execução do programa. Por isso, objetos são ditos monomórficos.

Por outro lado, referências a objetos podem ser polimórficas, porque, embora o tipo estático de uma referência seja fixada no momento de sua declaração, seu tipo dinâmico, como o próprio nome indica, pode variar durante a execução do programa.

Esse polimorfismo decorre do mecanismo de herança, que permite a especialização de classes, criando uma relação **é-um** entre

objetos, de forma a que objetos da classe especializada possam ser usados onde os de sua superclasse forem esperados.

Um exemplo de polimorfismo referencial universal em Java é mostrado no trecho de programa a seguir, onde, na linha 7, um objeto do tipo `B` é usado no lugar de um objeto do tipo `A`.

```
1  class A {...}
2  class B extends A {...}
3  class C {
4     void g() {
5        A a = new A();
6        B b = new B();
7        a = b;
8        ...
9     }
10 }
11 ...
```

6.2 Polimorfismo funcional

Funções polimórficas são um conceito presente em praticamente todas linguagens de programação de alto nível, e pode ser considerado parte essencial do modelo de execução das linguagens orientadas por objetos.

O polimorfismo funcional universal, apresentado na Fig. 6.1, trata do caso de uma mesma função ser aplicável a parâmetros de diferentes tipos. Isto é, quando um mesmo corpo de função é capaz de atuar com diferentes tipos de parâmetros. Por isso, essas funções são também chamadas de polivalentes. Com base no conjunto e tipo de parâmetros aceitáveis pela função, o polimorfismo nessa categoria pode ser paramétrico ou de inclusão.

Considera-se dotada de polimorfismo funcional universal paramétrico a função que aceita parâmetros de qualquer um dos tipos

pertencentes a um universo predefinido, o qual varia de linguagem para linguagem. Um exemplo é a função `printf` da linguagem **C**, que aceita como parâmetro valores de qualquer um dos diversos tipos que podem ser impressos.

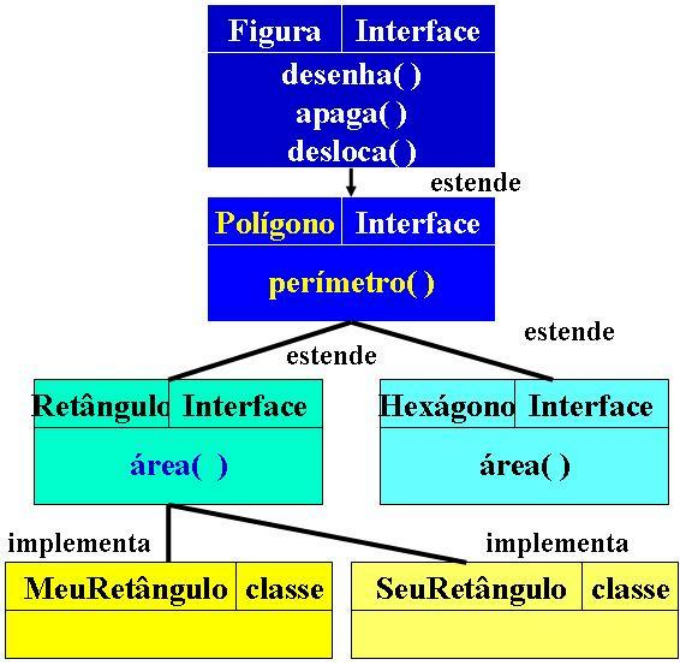


Figura 6.2 Hierarquia de Figuras

O polimorfismo funcional universal de inclusão decorre do entendimento de que uma função que pode receber parâmetros polimórficos é dita polimórfica. Como toda referência em Java é polimórfica, toda função que tem pelo menos um parâmetro cujo tipo é uma referência é polimórfica. Nesse caso, os parâmetros aceitáveis pela função são apenas aqueles cujos tipos estejam na hierarquia dos tipos em que os tipos dos respectivos parâmetros formais sejam suas raízes.

Por exemplo, considere a hierarquia de tipos especificada na Fig. 6.2, onde as operações definidas em cada tipo estão indicadas. Considere também a classe `Polimórficos`, definida a seguir, onde a hierarquia da Fig. 6.2 é usada para criar funções polimórficas de inclusão.

```
1  public class Polimórficos {
2     void f1(Figura x) { ... x.desenha(); ... }
3     void f2(Polígono y) {
4        ... y.desenha(); ... y.perímetro(); ...
5     }
6     void f3(Retângulo z) {
7        ... z.desenha(); ... z.área(); ...
8     }
9     void f4(MeuRetângulo w) {
10       ... w.desenha(); ... w.área(); ...
11    }
12 }
```

As funções `f1`, `f2`, `f3` e `f4` definidas acima são todas polimórficas de inclusão e, portanto, polivalentes, pois são aplicáveis a qualquer parâmetro cujo tipo dinâmico seja compatível com o especificado em cada caso.

No polimorfismo funcional ad-hoc, funções independentes podem ser homônimas, sendo que cada definição associada ao nome da função não precisa ter qualquer relação com outras definições já existentes, ou então há apenas uma função associada ao nome, mas essa função aceita parâmetros oriundos de um conjunto específico e limitado de tipos que podem ser convertidos para o tipo do parâmetro formal correspondente.

A esses dois tipos de polimorfismo dão-se os nomes de polimorfismo funcional de sobrecarga ou polimorfismo funcional de coerção.

Assim, ou tem-se que o número de novos significados é finito e explicitamente definido caso a caso ou as conversões possíveis também são predefinidas. Por essa razão, esse tipo de polimorfismo é chamado de ad-hoc .

Na classe `SobreCarga` definida a seguir, o nome `pesquisa` denota duas funções distintas, as quais são polimórficas de sobrecarga.

```
1  class SobreCarga {
2     Pessoa p = new Pessoa();
3     int[] tab;
4     Pessoa[] cadastro;
5     boolean b;
6     ...
7     boolean pesquisa(int[] tabela, int x) {...}
8     boolean pesquisa(Pessoa[] quadro, Pessoa p) {...}
9     void g() {
10      b = pesquisa(tab,100);
11      ...
12      b = pesquisa(cadastro,p);
13      ...
14    }
15 }
```

Os tipos dos parâmetros das chamadas `pesquisa(tab,100)` e `pesquisa(cadastro,p)` permitem ao compilador Java identificar a função a ser chamada em cada caso.

O nome de uma função polimórfica de sobrecarga é dito polissêmico, pois tem vários significados, que são referentes às funções que ele denota, ou seja, mais de um significado é associado a um mesmo nome.

Note que na polivalência, conceito ligado a polimorfismo de inclusão, cada função tem um único significado, mas tem a capacidade de operar com parâmetros polimórficos, portanto de diferentes tipos. Essas funções polimórficas são polivalentes, no sentido de que têm definições únicas que são eficazes para parâmetros de diferentes tipos.

Ressalte-se que o polimorfismo de funções pode manifestar-se simultaneamente nas duas dimensões, i.e., funções podem ser ao mesmo tempo polissêmicas e polivalentes.

O último tipo de polimorfismo da classificação original de Car-

delli & Wagner, e preservada na Fig. 6.1, o funcional ad-hoc de coerção, é o que decorre da possibilidade de conversão automática de tipo, comum em muitas linguagens de programação para simplificar a escrita de programas.

O exemplo mais comum são funções definidas com parâmetros formais de tipo numérico, e.g., `float sin(float x)`, e que aceitam automaticamente parâmetros de chamada de outros tipos de numéricos, como `int`.

Nesses casos, os parâmetros de chamadas são automaticamente convertidos para o tipo esperado. A esse tipo de conversão automática de valores dá-se o nome de coerção.

6.3 Resolução de sobrecarga estática

Polissemia ocorre quando um mesmo nome de função é usado para designar funções distintas e possivelmente não-relacionadas. No exemplo a seguir, as classes `A` e `B` implementam seis funções distintas com o mesmo nome `f`:

```
1  class A {
2     public void f(T x) {...}
3     public W f(T x, U y) {...}
4     public R f() {...}
5  }
6  class B {
7     public void f(T x) {...}
8     public int f(T x, U y) {...}
9     public float f() {...}
10 }
```

Os significados de cada ocorrência do nome `f` no programa abaixo são identificáveis a partir dos tipos estáticos de seus argumentos em cada invocação.

```
1  public UsoDeSobreCarga1 {
2     public static void main(String[] args) {
3        T t = new T();
4        U u = new U();
5        W c;  R m;
6        A a = new A();
7        A b = new B();
8        a.f(t);          // função f da linha 2 de A
9        c = a.f(t,u);    // função f da linha 3 de A
10       m = a.f();       // função f da linha 4 de A
11       b.f(t);          // função f da linha 7 de B
12       c = b.f(t,u);    // função f da linha 8 de B
13       m = b.f();       // função f da linha 9 de B
14    }
15 }
```

Em Java, o tipo de retorno de métodos não é usado para diferenciar operações polissêmicas, mas o tipo do objeto receptor é devidamente utilizado nesse processo, como ilustram as chamadas de `f` das linhas 10 e 11.

Na presença de hierarquia de classes, o mecanismo de resolução estática de sobrecarga de funções torna-se mais elaborado. A identificação da implementação do método polissêmico a ser executada ainda é determinada a partir dos tipos dos parâmetros de chamada, mas pode haver mais de um candidato que satisfaça o critério de identificação.

Nesses casos, a relação hierárquica entre o tipo de cada parâmetro formal e o tipo do argumento correspondente deve ser também usada para desambiguar, dando-se preferência à função cujo tipo de parâmetro formal em análise seja mais próximo hierarquicamente do tipo do parâmetro de chamada correspondente.

Para ilustrar essas novas regras, considere a seguinte hierarquia de classes:

```
1 public class A {int x; ... }
2 public class B extends A {int y; ... }
3 public class C extends B {int z; ... }
```

O programa `Teste1`, apresentado a seguir, imprime o texto `"f(A)"`, pois, embora o tipo dinâmico de `x` seja `C`, o seu tipo estático é `A`, o que causa a escolha da função `f` da linha 6, cujo parâmetro tem exatamente o mesmo tipo.

```
1 public class Teste1 {
2    public static void main(String[] args) {
3       A x = new C();
4       f(x); //Imprime "f(A)"
5    }
6    public static void f(A a) {System.out.println("f(A)");}
7    public static void f(B b) {System.out.println("f(B)");}
8 }
```

No exemplo `Teste2`, a escolha recai sobre o `f` da linha 7, porque agora o tipo estático do argumento passado `y` é `B`.

```
1 public class Teste2 {
2    public static void main(String[] args) {
3       B y = new C();
4       f(y); //Imprime "f(B)"
5    }
6    public static void f(A a) {System.out.println("f(A)");}
7    public static void f(B b) {System.out.println("f(B)");}
8 }
```

No caso do programa `Teste3`, a situação é diferente, pois o tipo estático do argumento da chamada `f(z)` da linha 4 não casa exatamente com o tipo do parâmetro formal da função definida na linha 6 nem com o da função da linha 7, mas ambas funções são aplicáveis, porque o tipo do parâmetro de chamada, `C`, é compatível

com os tipos `A` e `B` dos respectivos parâmetros formais das funções `f` das linhas 6 e 7.

```
1 public class Teste3 {
2    public static void main(String[] args) {
3       C z = new C();
4       f(z); //Imprime "f(B)"
5    }
6    public static void f(A a) {System.out.println("f(A)");}
7    public static void f(B b) {System.out.println("f(B)");}
8 }
```

Nesse caso, a função escolhida é aquela cujo parâmetro tenha um tipo que seja mais próximo, na hierarquia de tipo, do tipo do argumento, ou seja, a função `f` definida na linha 7 deve ser escolhida, pois na hierarquia de classes `C` $\rightarrow$ `B` $\rightarrow$ `A`, onde `A` é a raiz, `C` está mais próximo de `B` do que de `A`.

Quando as funções sobrecarregadas têm mais de um parâmetro, as regras acima devem ser aplicadas a todos os parâmetros, e a conjunção dos canditados encontrados deve conter uma única função.

```
1  public class Teste4 {
2     public static void main(String[] args) {
3        A a = new A();
4        B b = new B();
5        f(a,b); //Imprime "f(A,B)"
6        f(b,a); //Imprime "f(B,A)"
7     }
8     public static void f(A a, B b) {
9        System.out.println("f(A,B)");
10    }
11    public static void f(B b, A a) {
12       System.out.println("f(B,A)");
13    }
14 }
```

No exemplo acima, a chamada `f(a,b)`, da linha 5 da classe `Teste4`, aciona a função definida na linha 8, por ser essa é a única possibilidade, pois o tipo do primeiro argumento de chamada é incompatível com o correspondente da função da linha 11.

Raciocínio semelhante mostra que a chamada `f(b,a)` da linha 6 leva à execução da função definida na linha 11.

Entretanto, no próximo exemplo, o mecanismo de resolução de sobrecarga estática descrito não funciona. Note que as chamadas das linhas 7 e 8 da classe `Teste5` devem ser rejeitadas pelo compilador Java por serem ambíguas, pois as duas funções, as das linhas 10 e 13, são elegíveis.

```
1  public class Teste5 {
2     public static void main(String[] args) {
3        A a = new A();
4        B b = new B();
5        C c = new C();
6        f(a,b); f(b,a); f(a,c);
7        f(b,c); //Erro de compilacao
8        f(c,c); //Erro de compilacao
9     }
10    public static void f(B b, A a) {
11       System.out.println("f(B,A)");
12    }
13    public static void f(A a, B b) {
14       System.out.println("f(A,B)");
15    }
16 }
```

Esse tipo de ambiguidade pode ocorrer quando se tem mais de um parâmetro, pois o mecanismo, sendo aplicado a cada parâmetro de forma independente, pode produzir um resultado final com mais de uma função candidata.

No caso em questão, nas chamadas das linhas 7 e 8, há duas pos-

sibilidades de escolha, que a regra de proximidade de tipo não resolve, pois `f(b,c)` é compatível com a função `void f(B b,A a)` e também compatível com a função `void f(A a,B b)`. Pelo critério de proximidade, parâmetro de chamada `b` sugere que se escolha a primeira função, pois o tipo de `b` está mais próximo de `B` de que de `A`. Entretanto, o parâmetro `c` sugere a escolha da segunda função, pois `C`, o tipo do parâmetro formal, é está mais próximo de `B` do que de `A`. Essa ambiguidade gera o erro de compilação.

6.4 Resolução de sobrecarga dinâmica

A redefinição de um método no processo de especialização de classes sobrecarrega o nome do método com uma nova definição, acrescentando um novo significado a esse nome ao longo da hierarquia de classes que contêm o método. Nesses casos, as chamadas de métodos de mesmo nome não são distinguíveis em tempo de compilação pelo tipo e número dos parâmetros de chamada, porque todos têm exatamente a mesma assinatura, exceto pelo tipo do objeto receptor, cujo tipo dinâmico somente é conhecido em tempo de execução. Assim, é exatamente o tipo dinâmico do objeto receptor, que é considerado o primeiro parâmetro do método chamado, que vai determinar a função a ser ativada.

E como tipo dinâmico de uma referência normalmente não é determinável durante a compilação, a ligação entre a chamada do método e a sua versão a ser executada somente pode ser feita durante a execução. Desta forma, o compilador deve gerar um código apropriado para resolver esse tipo de sobrecarga durante a execução, conforme descreve a Seção 4.4.1, que trata de tabelas de métodos virtuais.

Considere a seguinte hierarquia das classes `A`, `B` e `C` e as diversas

definições das funções polimórficas `f` e `g`.

```
1  class A {
2     public void f(T d) {...}
3     public void g(U e) {...}
4  }
5  class B extends A {
6     public void f(T d) {...}
7  }
8  class C extends B {
9     public void f(T d) {...}
10    public void g(U e) {...}
11 }
```

O programa `SobreCarga2`, a seguir, mostra os detalhes da resolução dinâmica da sobrecarga. Note que a função identificada pelo mecanismo de resolução de sobrecarga está no comentário associado a cada chamada. As linhas mencionadas referem-se às classes `A`, `B` e `C` definidas acima.

```
1  class SobreCarga2 {
2    public static void main(String[] args) {
3        T t = new T();
4        U u = new U();
5        A a = new A();
6        A b = new B();
7        A c = new C();
8        a.f(t);         // função f da linha 2
9        a.g(u);         // função g da linha 3
10       b.f(t);         // função f da linha 6
11       b.g(u);         // função g da linha 3
12       c.f(t);         // função f da linha 9
13       c.g(u);         // função g da linha 10
14    }
15 }
```

Observe que a identificação da versão do método a ser invocado é feito em três etapas:

1. Inicialmente, a referência ao objeto receptor de cada invocação de um método é considerada como um parâmetro adicional cujo tipo é o tipo estático da referência ao objeto receptor.
2. A seguir, o compilador usa os tipos estáticos dos argumentos especificados, incluindo o da referência ao objeto receptor, para determinar o conjunto dos métodos candidatos.
3. Depois o compilador gera um código para, durante a execução, seja feita a escolha do método, dentre os candidatos, a ser ativado, com base no tipo dinâmico do objeto receptor.

Enfatiza-se que tipos estáticos são determináveis pelo compilador, mas a resolução da sobrecarga de métodos pelo tipo dinâmico somente pode ser feita durante a execução.

6.5 Reúso de funções

Polimorfismo está diretamente ligado a reúso de componentes de software. Funções polimórficas são mais reusáveis que funções monomórficas. Módulos que usam funções polimórficas são mais independentes de contexto, e, portanto, mais reusáveis.

O polimorfismo de sobrecarga permite simplificar o código do módulo cliente, porque elimina a necessidade de se identificar explicitamente o serviço a ser solicitado em cada caso. A identificação do corpo da função cujo nome é sobrecarregado é automaticamente feita ou pelo compilador ou pelo sistema de execução.

Funções polimórficas polivalentes evitam duplicação de código, porque são capazes de, com uma única definição, processar parâmetros de diferentes tipos. Em linguagens que não permitem

definição de funções polivalentes, o código dessas funções deveria ser duplicado e adaptado sempre que fossem introduzidos novos tipos de parâmetros no programa.

Polimorfismo de polivalência é uma consequência do mecanismo de herança de classes, pois polivalência depende diretamente das hierarquias de tipos construídas no programa: quanto mais profunda for uma hierarquia, maior é o grau de polivalência das funções que têm como parâmetros referências a tipos dessa hierarquia.

6.6 Reúso com interfaces

Herança múltipla de classes, que é um mecanismo de reúso não permitido em Java, cria mais oportunidades de polimorfismo de inclusão do que herança simples. Há situações, como a mostrada na Fig. 6.3, em que herança múltipla surge como uma solução natural e direta para a modelagem do problema.

A proibição de herança múltipla de classes em Java simplifica o sistema de tipo da linguagem e facilita a implementação de seu compilador, mas limita as oportunidades de polimorfismo que se poderiam ter, causando pequena perda no poder de expressão da linguagem. Sem herança múltipla, certas funções podem ter que ser duplicadas para cobrir toda a funcionalidade associada aos seus nomes. Entretanto, essa perda de poder de expressão pode ser aliviada pelo uso de interfaces.

Para ilustrar como essa perda de poder de expressão ocorre e é solucionada, suponha que herança múltipla de classe fosse permitida em Java, e considere a modelagem apresentada nas declarações abaixo e resumida na hierarquia mostrada na Fig. 6.3, na qual os diagramas apresentam no seu lado esquerdo os atributos privados da classe representada, e do lado direito, as suas operações públicas.

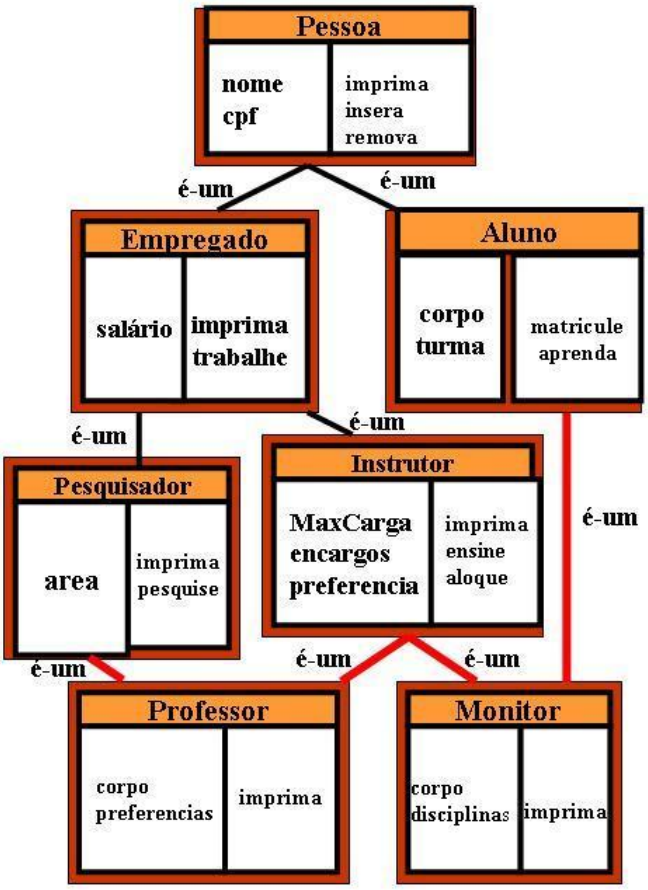


Figura 6.3 Sistema Acadêmico com Herança Múltipla

As classes da Fig. 6.3 são definidas a seguir. A classe `Pessoa` centraliza os elementos comuns a todas as demais na hierarquia, pois `Alunos`, `Pesquisadores` e `Instrutores` são `Pessoas`. E processos que aceitam `Pessoas` devem aceitar qualquer um deles. Suponha que a classe `Dados` esteja convenientemente definida.

```
1 class Pessoa {
2    private String nome;
3    private Dados dados;
4    public  Pessoa(String nome) { ... }
5    public  void  imprima() { ... }
6    public  void  defineDados(Dados d) { ... }
7    public  Dados obtémDados() { ... }
8 }
```

Classes **Aluno** e **Empregado**, definidas a seguir, são especializações da classe **Pessoa**.

```
1  class Aluno extends Pessoa {
2    private static LinkedList corpo;
3    private int número;
4    private Disciplina[] disciplinas;
5    public Aluno(String nome, int número) {super(nome);...}
6    public void matricule(Disciplina disciplina) {...}
7    public void aprenda(...) { ... }
8  }
9  class Empregado extends Pessoa {
10   private float salario;
11   private String cpf;
12   public Empregado(String nome,String cpf,float salário) {
13      super(nome);
14      this.cpf = cpf;
15      ...
16   }
17   public void imprima() { ... }
18   public void trabalhe() { ... }
19 }
```

E ainda há dois tipos de empregados: os pesquisadores e os instrutores, sendo estes encarregados das aulas da Instituição. Por serem empregados, suas classes devem estender a classe **Empregado**.

```
1  class Pesquisador extends Empregado {
2     private String área;
3     public Pesquisador(String nome, String área,
4                        String cpf,float salário) {
5        super(nome, cpf, salário);
6        ...
7     }
8     public void pesquise(Tema tema) { ... }
9  }
```

E os instrutores são definidos pela seguinte classe, que difere bastante da classe `Pesquisador`:

```
class Instrutor extends Empregado {
   private int MaxCarga = 2;
   private Disciplinas[] encargos;
   private Disciplinas[] preferências;
   public Instrutor(String nome, String cpf,
                            float salário, ...) {
      super(nome,cpf,salário);  ...
   }
   public void imprima() { ... }
   public void ensina() { ... }
   public void aloque(Disciplinas[] encargos) {...}
}
```

Há também professores, que, além de lecionar, fazem pesquisas, sendo, portanto, tipos especiais da classe `Pesquisador` e da classe `Instrutor`. A implementação natural da classe `Professor` seria uma que estendesse simultaneamente `Pequisador` e `Instrutor`.

Entretanto, Java não permite isso. Para fins de argumentação, suponha que Java permitisse herança múltipla, e assim, definir-se-ia a seguir a classe `Professor` com subtipo de `Pesquisador` e a de `Instrutor`, onde detalhes das chamadas das construtoras das superclasses foram omitidas para não obscurecer o exemplo.

```
class Professor extends Pesquisador, Instrutor {
   private static LinkeList corpo;
   public Professor(String nome, String área,
                    String cpf, float salário) {
      super(...);   ...
   }
   public void imprima() { ... }
}
```

Como monitores são alunos que exercem funções de instrutores, herança múltipla surge novamente como uma solução natural. Assim, define-se `Monitor` como subtipo de `Aluno` e de `Instrutor`:

```
1 class Monitor extends Aluno, Instrutor {
2    private static LinkedList corpo;
3    private Disciplinas[] disciplinas;
4    public Monitor(String nome, String área,
5                   String número) {super(...); ...}
6    public void imprima() { ... }
7 }
```

Note que os métodos de nomes `consulte`, `contrate` e `avalie` da classe `Aplicação1` são polimórficos polivalentes: uma única implementação de cada um deles é usada com argumentos de mais de um tipo. Por exemplo, na linha 11, duas chamadas ao mesmo método `consulte`, definido na linha 2, são realizadas, sendo uma para referência a `Pesquisador`, e a outra, a `Professor`.

```
1  public class Aplicação1 {
2     void consulte(Pesquisador x) { ... }
3     void avalie(Aluno z) { ... }
4     void contrate(Instrutor y) { ... }
5     public static void main(String[] arg) {
6        Pesquisador q = new Pesquisador();
7        Professor   p = new Professor();
8        Instrutor   i = new Instrutor();
9        Aluno       a = new Aluno();
10       Monitor     m = new Monitor();
11       consulte(q); consulte(p);
12       contrate(p); contrate(m);
13       avalie(a);   avalie(m);
14       ...
15    }
16 }
```

Os benefícios do polimorfismo gerado pelo uso de herança simples e múltipla na modelagem acima revelam-se em termos do alto grau de reúso que se observa na aplicação apresentada acima.

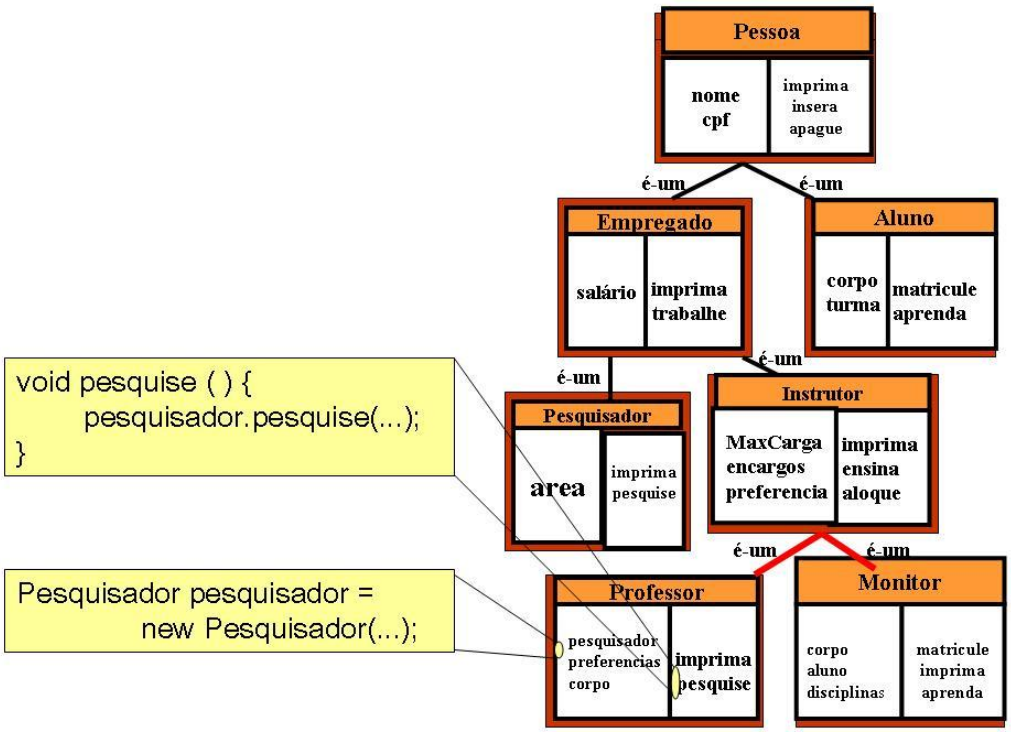


Figura 6.4 Sistema Acadêmico sem Herança Múltipla

6.6.1 Uso de herança simples

As duas ocorrências de herança múltipla devem ser eliminadas do exemplo apresentado nesta seção para que sua implementação em Java torne-se válida. Recomenda-se a quebra das relações **é-um** que causem menor perda de reúso.

Em geral, é preferível a quebra da relação que interromper o caminho mais curto até à raiz da hierarquia. No caso em análise, as relações `Professor-Pesquisador` e `Monitor-Aluno` são as escolhidas para ser eliminadas, conforme mostram Fig. 6.3 e Fig. 6.4.

Com a eliminação da relação entre `Professor` e `Pesquisador` e entre `Monitor` e `Aluno`, os métodos `consulte` e `avalie` perdem

polivalência e devem ser duplicados para conservar a funcionalidade, como mostra a implementação de `Aplicação2`.

```
1  public class Aplicação2 {
2     void consulte(Pesquisador x) { ... }
3     void consulte(Professor x) { ... }
4     void avalie(Aluno z) { ... }
5     void avalie(Monitor z) { ... }
6     void contrate(Instrutor y) { ... }
7     public static void main(String[] arg) {
8        Pesquisador q = new Pesquisador();
9        Professor   p = new Professor();
10       Instrutor   i = new Instrutor();
11       Aluno       a = new Aluno();
12       Monitor     m = new Monitor();
13       consulte(q); consulte(p);
14       contrate(p); contrate(m);
15       avalie(a);   avalie(m);
16       ...
17    }
18 }
```

Perde-se também, nesse processo, reúso de implementação. Campos e métodos de `Pesquisador` não são mais herdados pela nova classe `Professor`, que deve defini-los diretamente.

A replicação dos campos antes herdados da classe `Pesquisador` pode ser realizada criando-se em `Professor` o campo

```
Pesquisador pesquisador = new Pesquisador()
```

para agregar aos professores seu atributo pesquisador.

A implementação do método `pesquise`, não mais herdado de `Pesquisador`, pode ser feita na classe `Professor` por meio de delegação:

```
void pesquise (...) { pesquisador.pesquise(...); }
```

Similar perda de reúso ocorre na eliminação da relação **é-um** entre `Monitor` e `Aluno`.

Em suma, a maior perda de reúso reside na necessidade de duplicação dos métodos `consulte` e `avalie`. Entretanto, essas indesejáveis duplicações podem ser minimizadas pelo uso de interfaces.

6.6.2 Herança múltipla com interfaces

Interfaces também podem ser usadas para restabelecer a perda de polivalência das funções `consulte` e `avalie`, descrita no exemplo acima, e evitar duplicação de código. Uma solução é colocar as classes `Professor` e `Pesquisador` em uma mesma hierarquia aceitável pelo método `consulte`. E analogamente o mesmo deve ser feito com `Monitor` e `Aluno` para o método `avalie`. Para isso, deve-se inicialmente criar as seguintes interfaces:

```
1 interface PesquisadorI {
2    void pesquise(Tema tema);
3    void imprima();
4 }
5 interface AlunoI {
6    void matricule();
7    void aprenda();
8 }
```

A seguir altera-se a hierarquia de tipos, modificando as declarações de `Pesquisador`, `Professor`, `Aluno` e `Monitor`.

```
1 class Pesquisador extends Empregado
2                             implements PesquisadorI {...}
3 class Professor extends Empregado
4                             implements PesquisadorI {...}
5 class Aluno extends Pessoa implements AlunoI {  ... }
6 class Monitor extends Empregado implements AlunoI {  ... }
```

Os corpos das classes acima não foram alterados em relação a versão anterior, que usa apenas herança simples. A nova hierarquia está na Fig. 6.5.

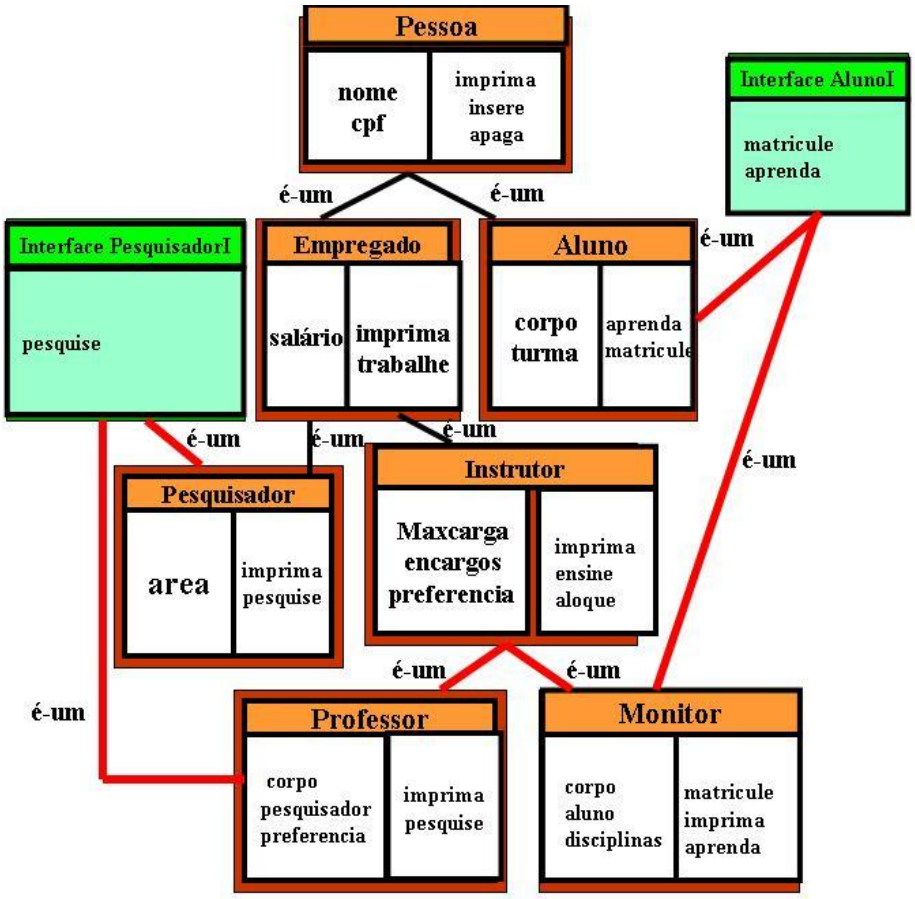


Figura 6.5 Sistema Acadêmico com Interfaces

Por fim, mudam-se os tipos dos parâmetros de `consulte` e `avalie`, passando-se a ter os métodos

```
void consulte(PesquisadorI x) { ... }
void avalie(AlunoI z) { ... }
```

no lugar dos quatro métodos definidos da linha 2 à linha 5 da classe `Aplicação2`. Note que os corpos desses métodos não precisam ser modificados. Apenas os tipos dos parâmetros é que devem ser alterados.

Claramente o recurso de interface permite uma modelagem quase tão natural quanto à oferecida pelo mecanismo de herança múltipla, justificando a decisão de simplificar a implementação do compilador Java pela opção de permitir apenas herança simples para classes.

6.7 Paradoxo da herança

Especialização é essencialmente um processo de se adicionar novas características a um tipo de dados, criando-se assim um novo tipo com propriedades particulares.

Se para cada tipo existe um conjunto subjacente de valores dos objetos do tipo, a adição de novas características a esses objetos define subconjuntos desses valores.

A especialização é, portanto, um processo de descrever subconjuntos. Por exemplo, considere a classe `Médico` e sua subclasse `Cirurgião`, definidas da seguinte forma:

```
1  class Médico {
2    "atributos de dados privados"
3    public Diagnóstico consulte(Paciente p) { ... }
4    public Remédio receite(Paciente p, Diagnóstico d) {...}
5  }
6  class Cirurgião extends Médico {
7    "atributos de dados privados"
8    public Diagnóstico consulte(Paciente p) {...}
9    public void opere(Paciente p, Diagnóstico d) {...}
10 }
```

A classe `Médico` descreve o conjunto de todos os médicos, os quais são caracterizados pelas operações indicadas. A especialidade médica `Cirurgião`, que reúne um subconjunto dos médicos, é definida com um subtipo de `Médico` pela agregação de uma nova operação e redefinição da implementação de uma das que foram herdadas. São essas operações que tornam o `Cirurgião` um tipo especial de `Médico`.

Observe que, no programa `Teste` abaixo, referências declaradas com o tipo estático `Médico` podem também apontar para objetos da subclasse `Cirurgião`, como ilustrado na linha 6, e que subclasse

(ou subtipo) corresponde a subconjunto, enquanto classe (ou tipo) descreve o conjunto associado.

```
1 public class Teste {
2   public static void main(String[] args) {
3      Médico m = new Médico();
4      Cirurgião c = new Cirurgião ();
5      ...
6      m = c;
7      ...
8   }
9 }
```

Assim, tomando como base os conjuntos subjacentes aos tipos, diz-se que subtipos estão incluídos nos respectivos tipos. E devido a esse fato, isto é, o de subtipos estarem incluídos nos respectivos tipos, como mostra o relacionamento **é-um** existente entre `Cirurgião` e `Médico`, referências são ditas dotadas de polimorfismo de inclusão.

No exemplo `Médico-Cirurgião`, herança é usada para criar hierarquia de tipos e subtipos. O objetivo é dar início à classificação dos diversos tipos de médicos em categorias hierárquicas ligadas pela relação **é-um** e obter os benefícios de reúso concedidos pelo polimorfismo de inclusão das referências aos objetos associados a essa hierarquia.

Entretanto, herança também pode ser usada com o objetivo de se obter reúso de implementação. Nesses casos, conceitualmente subtipos não são criados, embora na prática subclasses ainda possam ser tratadas como subtipos, e o polimorfismo das referências a objetos das classes envolvidas ainda continua válido. Porém, paradoxalmente, nesses casos, as correspondências conjunto-tipo e subconjunto-subtipo não são mais válidas.

Para se compreender esse fenômeno, considere que um ponto

`(x,y)` no sistema cartesiano possa ser descrito pela seguinte classe `Ponto`:

```
1  public class Ponto {
2     private float x1, y1;
3     public Ponto(float a, float b) {x1=a; y1=b;}
4     protected void ajuste(float x, float y) {...}
5     protected void apague() {...}
6     protected void exiba() {...}
7     public void mova(float x , float y ) {
8        apague();
9        ajuste(x,y);
10       exiba();
11    }
12 }
```

Observe que a operação `mova`, para deslocar um ponto de um lugar para outro, faz uso de três funções internas para apagar o ponto, calcular suas novas coordenadas e re-exibi-lo no novo local.

Suponha que essa sequência de ações seja indispensável para uma boa implementação de `mova` e que essas operações internas sejam específicas para o objeto `Ponto` associado, e não necessariamente se aplicam à movimentação de outros tipos de objetos. Por isso, elas foram definidas com visibilidade `protected` para que especializações de `Ponto` possam redefini-las adequadamente.

Certamente, em uma aplicação real, a classe `Ponto` deveria ter outras operações, mas, para ilustrar a questão do uso de herança puramente com o objetivo de reúso de implementação, apenas a operação `mova` é suficiente.

Considerando-se que um segmento de uma reta seja definido por dois pontos, pode-se definir a classe `Segmento` abaixo como uma especializacão de `Ponto`, que adiciona mais um par de coordenadas e redefine apropriadamente as operações herdadas de `Ponto`, conforme mostra-se a seguir.

```
1  public class Segmento extends Ponto {
2    private float x2, y2;
3    public Segmento(float a, float b, float c, float d) {
4       super(a,b); x2 = c; y2 = d;
5    }
6    protected void ajuste(float x, float y) {...}
7    protected void apague() {...}
8    protected void exiba() {...}
9    public int comprimento() {...}
10 }
```

A implementação do método **mova** de **Ponto**, herdada pela subclasse **Segmento**, foi totalmente reusada, embora sua semântica tenha sido apropriadamente adaptada pela redefinição dos métodos herdados **ajuste**, **apague** e **exiba**.

Observe que no programa **Figura** abaixo, o parâmetro **z** do método **h** é polimórfico, haja vista que durante a execução de **h**, ativada pela chamada da linha 6, o tipo dinâmico de **z** é **Ponto**, enquanto na execução de **h** ativada pela linha 8, o tipo é **Segmento**.

```
1  public class Figura {
2    public void h(Ponto z) {...; z.mova(20.0,20.0); ... }
3     public void g() {
4        Ponto  p; Segmento r;
5        p = new Ponto(10.0, 10.0);
6        h(p);
7        r = new Segmento(5.0, 5.0, 15.0, 15.0);
8        h(r);
9        ...
10    }
11    public static void main(...) {
12       Figura f; ... f.g(); ...
13    }
14 }
```

O caráter polimórfico da referência `z` permite à operação `mova` utilizar a versão correta dos métodos `ajuste`, `apague` e `exiba`, conforme o tipo do objeto passado em cada chamada, que é `Ponto` na primeira chamada, e `Segmento`, na segunda.

Tudo funciona bem, pois, pela definição apresentada, a classe `Segmento` é vista como subtipo de `Ponto`, e, portanto, um objeto do tipo `Segmento` pode ser usado onde um objeto `Ponto` for esperado. Entretanto, não é razoável considerar que o conjunto de objetos do tipo `Segmento` seja um subconjunto dos de tipo `Ponto`, contrariando a visão defendida no caso das classes `Médico` e `Cirurgião`.

Esse aparente paradoxo decorre da natureza dual do mecanismo de herança, cujo uso serve a dois propósitos bem distintos: criação de hierarquia de tipos e reúso de implementação. No primeiro caso, subclasses sempre correspondem a subconjuntos, como se espera em uma hierarquia de tipos, mas no segundo caso, essa relação pode não ser válida. De um ponto de vista metodológico, o uso de herança para simplesmente atingir reúso deveria ser evitado, dando-se preferência, nesse caso, à composição de objetos, que é também um mecanismo de reúso de código.

6.8 Conclusão

Polimorfismo é um conceito usado em linguagens de programação desde os primórdios da Computação, e.g., coerção existe desde a primeira versão do Fortran.

Mais tarde, com a invenção do conceito de tipo abstrato de dados e hierarquia de tipos, seu uso proliferou-se em praticamente todas as linguagens de programação, sendo hoje um recurso indispensável para efetivamente praticar reúso de código.

O polimorfismo de Java é bastante abrangente e cobre todas as principais nuances desse conceito.

Exercícios

1. Além de referências e funções, o que mais pode ser polimórfico em Java?
2. Que dificuldades de programação poderiam ser encontradas em uma linguagem orientada por objetos sem o recurso de referências ou apontadores?
3. Dê um exemplo de procedimentos da vida real, fora do mundo da Computação, em que polimorfismo é usado.
4. Dado que existe o tipo interface em Java, classes abstratas ainda são úteis?
5. Qual é a utilidade de ser ter tipos estáticos?
6. Qual é o principal benefício do mecanismo de ligação dinâmica?
7. Que são linguagens fortemente tipadas?

Notas bibliográficas

O termo polimorfismo, no contexto de linguagens de programação, foi introduzido por Cristhropher Strachey em 1967 [51, 52].

Strachey classificou o polimorfismo em *ad-hoc* e paramétrico . No polimorfismo *ad-hoc*, o número de tipos possíveis associados a um nome é finito e esses tipos devem ser individualmente definidos no programa. No paramétrico, as funções são escritas independentemente dos tipos dos parâmetros e podem ser usadas com qualquer novo tipo.

Em 1985, Luca Cardelli e Peter Wegner[4] modificaram a classificação de polimorfismo de Strachey, criando a categoria de polimorfismo universal para generalizar o paramétrico.

Nessa classificação, o polimorfismo que se manisfesta em linguagens de programação pode ser:

- *ad-hoc*:
  - sobrecarga
  - coerção
- universal:
  - paramétrico
  - inclusão

Capítulo 7

Tratamento de Falhas

Métodos de bibliotecas ou métodos servidores têm meios para detectar, durante sua execução, situações de erro que lhes impedem de cumprir sua tarefa, mas geralmente não sabem o que fazer nessas situações. Por outro lado, métodos clientes têm mais informação de contexto do que os servidores e frequentemente sabem o que fazer nos casos de falhas, mas geralmente não têm meios para detectar ou prever as situações de erro que podem ocorrer durante a execução do método servidor.

Uma solução para esse problema consiste em associar a detecção do erro a seu tratamento. Deve-se construir uma estrutura de programa que permita transmitir informações sobre os erros encontrados do ponto de sua detecção para os pontos em que podem ser tratados. Por exemplo, todo método servidor deveria, em vez de tentar tratar o erro encontrado, apenas retornar um código identificador do erro, e o cliente poderia testar o valor desse código após cada chamada ao método para tomar as providências necessárias. Entretanto, essa técnica tem o defeito de poluir o texto do programa com frequentes testes de validação de dados, que se devem realizar ao longo do programa, obscurecendo a sua semântica.

Essa indesejada poluição pode ser evitada pela programação de tratamento de erros por meio de lançamento e captura de exceções. Com o uso de exceções, métodos servidores passam a ter mais

de uma opção de retorno: retorno normal, imediatamente após o ponto de chamada, e retorno, via exceção, para outras posições bem definidas no programa. Servidores com mais de uma forma de retorno permitem aos métodos clientes perceberem, sem necessidade de testes, se houve ou não falha na execução do método servidor, e, assim, processar cada tipo de retorno de forma apropriada.

7.1 Declaração de exceções

Uma exceção em Java é representada por um objeto de um tipo da hierarquia da classe `Throwable`, mostrada na Fig. 7.1, onde destacam-se as classes `Error`, `Exception` e `RunTimeException`.

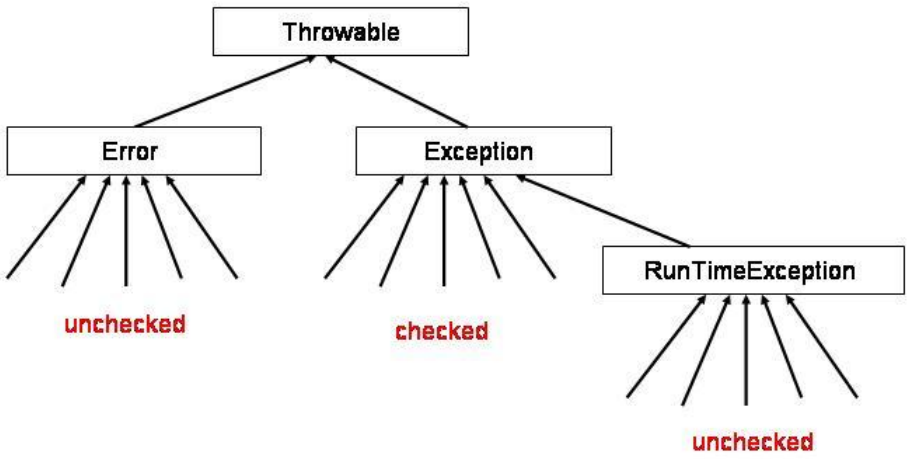


Figura 7.1 Hierarquia das Exceções

Para se criar uma classe de exceção, deve-se estender uma das classes da família de `Throwable`.

Exceções dos tipos `Error` e `RunTimeException` são chamadas exceções não-verificadas (*unchecked*). As extensões da classe `Exception` são exceções verificadas (*checked*). Exceções verificadas devem ser sempre anunciadas nos cabeçalhos dos métodos que podem lançá-las, mas não as tratam internamente. Todas as exceções, verificadas ou não, devem ser tratadas pela aplicação em

algum momento para evitar o risco de o programa ser abortado.

O cabeçalho de métodos que podem falhar, i.e., métodos que levantam exceções, mas não as tratam, delegando seu tratamento a métodos que os invocam, tem o formato:

```
<modificador> T f(lista de parametros)
                              throws <lista de exceções>
```

Um método sem a cláusula de `throws` não pode lançar exceções verificadas que não sejam por ele tratadas.

Cada nome na lista `throws` de um método especifica uma família de exceções: se um método declara que pode lançar uma exceção `A` em sua lista `throws`, ele pode também lançar qualquer descendente da classe `A`.

As exceções pré-definidas na linguagem Java são extensões das classes `Error` ou `RunTimeException`. Embora seja possível criar uma exceção estendendo qualquer uma das classes mostradas na Fig. 7.1, o usual nas aplicações é criá-las como exceção verificada, estendendo a classe `Exception`.

7.2 Lançamento de exceções

Um método que encontra uma situação de erro intransponível, que o impede de avançar na computação, deve reagir lançando uma exceção, via o comando `throw new X(...)`, o qual cria um objeto de exceção do tipo `X` e o lança, sendo `X` o nome de uma classe da família `Throwable`.

A exceção lançada é o objeto criado, que deve ser capturado por um `catch` para que o programa não seja abortado. Todo `catch` tem um único argumento, que é do tipo `throwable`. O objeto lançado por um comando `throw` pode ser capturado por um `catch` localizado no mesmo método em que o lançamento ocorre

ou em qualquer um dos métodos localizados no caminho inverso das chamadas ativas de métodos.

Tudo funciona como se o objeto lançado fosse um argumento passado remotamente à cláusula `catch` que trata a exceção. O tipo do objeto lançado identifica o `catch` que deve capturar e tratar a exceção.

Cada comando `try` delimita uma região do programa onde falhas são detectadas e define os `catches` que capturam e tratam falhas. O comando tem o formato descrito abaixo, onde as cláusulas `catch` e `finally` podem ser omitidas.

```
1 try {bloco de comandos b}
2 catch(Excecao1 e) {bloco de comandos b1}
3 catch(Excecao2 e) {bloco de comandos b2}
4 ...
5 catch(excecaoN e) {bloco de comandos bn}
6 finally {bloco de comandos f}
```

Um comando `try` funciona como um comando bloco. Quando ativado na sequência normal de execução, seu corpo, formado pelo bloco `b`, da linha 1, é executado na sequência do fluxo de controle e depois executa-se o bloco `f`, da linha 6, antes de se passar para o comando seguinte ao `try` no fluxo normal de execução. A cláusula `finally` é sempre executada, exceto em alguns casos especiais, discutidos na Seção 7.3.

Se durante a execução do bloco `b` alguma exceção for lançada, esse bloco é abandonado imediatamente, e inicia-se a busca por um `catch` para tratar a exceção. Inicialmente faz-se uma busca entre as cláusulas `catch`, linhas 2 a 5, que seguem o `try`, onde a exceção foi lançada. Seleciona-se a primeira cláusula cujo parâmetro formal tenha um tipo igual ou hierarquicamente superior ao tipo do objeto da exceção lançada. A referência ao objeto da exceção é associada ao parâmetro, e o controle segue no corpo da cláusula

escolhida. Terminada a execução dessa cláusula, o bloco `f`, da cláusula `finally`, é executado, e então o fluxo de controle passa para o comando seguinte ao `try`.

Caso a cláusula `catch` desejada não seja encontrada no comando `try`, o bloco `finally` associado é executado, e a busca continua no comando `try` envolvente, se houver. Do contrário, o método em execução é abandonado, e a busca por um `try` ativo prossegue no método chamador, onde retoma-se a busca da cláusula `catch` desejada no `try` ativo encontrado, seguindo os procedimentos descritos acima, como se a exceção tivesse ocorrido nesse último `try`.

Nos fragmentos de programas abaixo, suponha que todos os comandos `throw` e `try` estejam explicitamente indicados, ou seja, nas partes omitidas, indicadas por `"..."`, não haja ocorrências desses comandos. A classe `E` é uma classe de exceção, e o método `g` de `A` pode ou não lançar a exceção `E`, dependo de seu fluxo de execução. Como essa exceção não é tratada em `g`, a possibilidade de seu levantamento deve ser anunciada no seu cabeçalho para que o chamador de `g` possa tomar providências a respeito.

```
1 class E extends Exception {
2    public int v;
3    public E(int v) {this.v = v;}
4 }
5
6 class A {
7    void g() throws E {int k; ... ; throw new E(k); ....}
8    ...
9 }
```

O programa `B` abaixo inicia-se com a execução dos comandos a partir da linha 3, e, ao atingir a linha 5, inicia a execução do bloco associado ao `try`. A chamada `a.g()` leva à execução da linha 7 da classe `A` acima, onde o comando `throw` pode ser alcançado ou não,

dependendo da ação dos comandos que o precedem. Caso não seja, a execução de `g` é finalizada, o controle volta imediatamente após `a.g()` da linha 5 da classe `B`, e a execução do restante do corpo do `try` é concluída. O bloco `finally`, da linha 7, é executado concluindo-se o comando `try`, e o fluxo segue na linha 8.

```
1  public class B {
2     public static void main(String[] args) {
3        A a;
4        ...
5        try { ... ; a.g(); ... }
6        catch (E e) {... System.out.println(e.v); ...}
7        finally { ... }
8        ...
9     }
10 }
```

Por outro lado, se durante a execução da linha 7 da classe `A`, definida acima, o comando `throw` for executado, a exceção `E` é lançada, e passada ao parâmetro do `catch` da linha 6 da classe `B`, e a execução continua no bloco de comandos dessa linha. A seguir, o bloco do `finally`, da linha 7, é executado para concluir o comando `try`, e o fluxo segue normalmente na linha 8.

Para melhor compreender os mecanismos do levantamento de exceções, considere agora a seguinte classe de exceção `Indice`:

```
1  class Indice extends Exception {
2     public int i;
3     Indice(int i) {this.i = i;}
4  }
```

A implementação abaixo de `MeuVetor` prevê erros de indexação, que se detectados, causam o levantamento de exceções nas linhas 10 e 15, que devem ser tratados adequadamente pelos clientes dos métodos `atribui` e `elemento`.

```
1  class MeuVetor {
2     private int [] p;
3     private int tamanho;
4     public void MeuVetor(int n) {
5        p = new int[n]; tamanho = n;
6     }
7     public int elemento(int i) throws Indice {
8        if ( 0 <= i && i < tamanho )
9            return p[i] ;
10       throw new Indice(i);
11    }
12    public void atribui(int i, int v) throws Indice {
13       if ( 0 <= i && i < tamanho )
14           p[i] = v ;
15       throw new Indice(i);
16    }
17 }
```

A execução linha a linha do programa `C`, apresentado na próxima página, começa pelo método `main` da linha 21 e segue a ordem:

1. Executam-se linha 22 para declarar e criar um objeto do tipo `C` e a linha 23 para ativar um serviço de `C`.
2. A chamada de `c.f()` causa a execução das linhas 3 a 6, que imprime o texto `Ponto A`, e a seguir inicia a execução da linha 7 de `C`.
3. A chamada `g(x)` da linha 7 leva o controle para a linha 18, que chama o método `elemento` de `MeuVetor`, que causa o levantamento da exceção `Indice`.
4. A execução então continua no corpo do `catch` da linha 10, que imprime a mensagem `Indice Errado = 10`, e depois retorna à linha 8 para imprimir `Ponto B`.
5. O último comando da função `f`, linha 14, imprime o resultado `Valor de k = 0` e conclui-se a execução de `c.f()`.

6. Após o retorno à linha 24 de `main`, imprime-se `Chegou aqui`.

```
1  public class C {
2     void f() {
3        MeuVetor  x = new MeuVetor(5);
4        int k;
5        try {
6           System.out.print("Ponto A");
7           k = g(x);
8           System.out.print("Ponto B");
9        }
10       catch(Indice x) {
11          System.out.print("Indice Errado = " + x.i);
12          k = 0;
13       }
14       System.out.println("Valor de k = " + k);
15    }
16
17    int g(MeuVetor v) throws Indice {
18        return v.elemento(10);
19    }
20
21    public static void main(String[] args){
22       C c = new C();
23       c.f();
24       System.out.println("Chegou aqui");
25    }
26 }
```

O exemplo abaixo mostra uma implementação de uma outra classe, `SeuVetor`, com a previsão de duas condições de erros que podem ocorrer no momento da criação de objetos da classe ou na indexação de seus elementos.

O primeiro erro é o que ocorre quando o tamanho especificado para o vetor for negativo, sendo indicado pelo levantamento da exceção `Tamanho`. O outro erro surge da tentativa de recuperação

de elementos do vetor armazenado no objeto com índices fora de seus limites, sendo levantada a exceção **Indice**.

```
1  class Indice extends Exception {
2     public int i;
3     Indice(int i) {this.i = i;}
4  }
5
6  class Tamanho extends Exception {
7     public int t;
8     public Tamanho(int t) {this.t = t;}
9  }
10
11 class SeuVetor {
12    private int [] p; private int tamanho;
13    public SeuVetor(int n) throws Tamanho {
14       if ( n < 0 )
15           throw new Tamanho(n);
16       p = new int[n]; tamanho = n;
17    }
18    public int elemento(int i) throws Indice {
19       if (0 <= i && i < tamanho) return p[i];
20       throw new Indice(i);
21    }
22    public void atribui(int i, int v) throws Indide {
23       if (0 <= i && i < tamanho) p[i] = v ;
24       else throw new Indice(i);
25    }
26
27 }
```

O programa **Usuário** abaixo, que testa a classe **SeuVetor**, prevê o tratamento apenas da exceção **Tamanho**.

O tratamento da exceção **Indice** é delegado à JVM, o que pode causar o cancelamento da execução. É de bom alvitre que **main** nunca deixe de tratar todas as exceções.

```
1  public class Usuário {
2     int n = ... ; //suponha bons e maus valores para n
3     static void f() throws Indice, Tamanho {
4        SeuVetor x = new SeuVetor(n);
5        try {SeuVetor z = new SeuVetor(n); g(z);}
6        catch(Indice e) { }
7        catch(Tamanho e) { }
8        g(x);
9     }
10    static void g(SeuVetor v) throws Indice {
11       k = v.elemento(100);
12    }
13    public static void main(String[] args) throws Indice {
14       try {f();}
15       catch(Tamanho e) {
16          System.out.print("Tamanho = " + e.t);
17       }
18       System.out.print(" e tudo sob controle");
19    }
20 }
```

A execução do `main` da classe `Usuário` leva a ativação da linha 14 da função `f`, a qual chama `g(z)` na linha 5, o qual invoca `v.elemento(100)` na linha 11. Se o valor de `n`, definido na linha 2, for menor que `100`, haverá o levantamento de `Indice` na linha 20 da classe `SeuVetor`, que levará à execução do `catch` da linha 6 de `Usuário`.

Todo método que pode falhar na execução de seu objetivo deve sinalizar sua falha via exceção. Ressalte-se que iniciadores de campos, contidos nas declarações, não devem fazer chamadas a métodos que falham, porque não se têm como capturar essas exceções nas declarações. Similarmente, blocos de iniciação de variáveis estáticas não podem lançar exceções, mas podem capturá-las.

7.3 Cláusula `finally`

A claúsula **`finally`** de comandos **`try`** é sempre executada. Sua função é dar a oportunidade de garantir as condições invariantes do bloco do **`try`** correspondente. Para ilustrar o funcionamento da cláusula **`finally`**, considere inicialmente o seguinte programa:

```
1  public class Finally1 {
2    static public void main(String[] args) {
3        Final a = new Final();
4        a.f();
5        System.out.print("Acabou");
6    }
7  }
8  class Final {
9     public void f() {
10       int x = 10;
11       System.out.print("x = " + x);
12       try { x = 11;
13             System.out.print("x = " + x);
14             throw new Exception();
15       }
16       catch(Exception e) {
17            x = 12;
18            System.out.print("x = " + x);
19       }
20      finally {
21            x = 13;
22            System.out.print("x = " + x);
23      }
24    }
25 }
```

O **`main`** de **`Finally1`** causa o seguinte fluxo de execução:

1. Executa-se linha 4 de **`main`**, chamando a função **`f`**.

2. Executam-se linhas 10, 11 (imprime `x = 10`), 12, 13 (imprime `x = 11`), 14 (exceção lançada).
3. Executam-se as linhas 16, 17, 18 (imprime `x = 12`).
4. Executa a cláusula `finally` associada, passando o controle para linha 21, 22 (imprime `x = 13`), 23 e retorna à linha 5 de `main`, quando imprime `Acabou`.

O programa `Finally2` mostra que a clásula `finally` é executada até mesmo quando o bloco do `try` executa um `return`, o qual somente é efetivado após `finally` ser executado, conforme mostra `Finally2` ao imprimir `x = 10 x = 11 x = 13 Acabou`.

```
1  public class Finally2 {
2     static public void main(String[] args) {
3       Final a = new Final();  a.f();
4       System.out.print("Acabou");
5     }
6  }
7  class Final {
8     public void f() {
9       int x = 10; System.out.print("x = " + x);
10      try { x = 11; System.out.print("x = " + x); return;}
11      catch(Exception e) {
12          x = 12; System.out.print("x = " + x);
13      }
14      finally { x = 13; System.out.print("x = " + x); }
15      System.out.print("Aqui não passa");
16    }
17 }
```

O mesmo vale para comandos `return` que ocorrem dentro de `catch`, os quais têm a mesma semântica dos que ocorrem dentro de `try`: somente são concluídos após a execução da cláusula `finally`, ou seja, depois de o `finally` assegurar que as condições de finalização do `try` estão satisfeitas. Note que o operando do

return é avaliado antes de se executar o **finally**, como mostra o programa **Finally3** abaixo, que imprime **x = 11** e não **x = 111**.

```
1  public class Finally3 {
2     static public void main(String[] args) {
3       Final a = new Final(); int x = a.f();
4       System.out.println("x = " + x);
5     }
6  }
7  class Final {
8     int x = 0;
9     public int f() {
10      try { x = 1; throw new Exception();}
11      catch(Exception e) { x = x + 10; return x;}
12      finally { x = x + 100;}
13    }
14 }
```

Quando o comando **return** ocorre na cláusula **finally**, ele prevalece sobre o do corpo do **try** ou de **catch**, sendo obedecido imediatamente. No programa abaixo, o valor impresso é **x = 1000**.

```
1  public class Finally4 {
2     static public void main(String[] args) {
3        Final a = new Final(); int x = a.f();
4        System.out.print("x = " + x);
5     }
6  }
7  class Final {
8     public int f() {
9        int x = 10;
10       try {return x; }
11       catch(Exception e) {x = 100;}
12       finally {x = 1000; return x;}
13    }
14 }
```

Finalmente, a cláusula `finally` não é executada no caso de o bloco de `try` invocar a operação `System.exit(0)`, para encerrar o programa. O programa abaixo imprime `x = 10` e não `x = 11`, demonstrando esse fato:

```
1  public class Finally5 {
2     static public void main(String[] args) {
3        Final a = new Final(); a.f();
4        System.out.print("Acabou");
5     }
6  }
7  class Final {
8     int x = 10;
9     static public void f() {
10       System.out.println("x = " + x);
11       try { x = 13; System.exit(0); }
12       catch(Exception e) {
13          x = 12; System.out.println("x = " + x);
14       }
15       finally {x = 11; System.out.print("x = " + x);}
16    }
17 }
```

Para concluir, lembre-se que em programação é bastante comum construir certas entidades, utilizá-las e depois descartá-las. Existem comandos para *fazer* e comandos para *desfazer*. Muitas vezes no início do bloco principal de um `try`, há comandos para *fazer*, enquanto que os respectivos comandos para *desfazer* ficam localizados no fim do bloco. Isso pode apresentar um problema de consistência, porque, na presença de exceção, não se pode garantir que os comandos de *defazer* do fim do bloco do `try` sejam sempre executados.

A solução para esse problema é a cláusula `finally` do comando `try` que abriga um bloco de comandos que sempre será executado

após o término normal ou via exceção do bloco do `try`. Isso faz da cláusula `finally` o local ideal dos comandos do tipo *desfazer*.

7.4 Objeto de exceção

Classe de exceção é uma classe como qualquer outra, exceto pelo fato de obrigatoriamente pertencer a família `Throwable`. Uma classe de exceção pode conter livremente métodos e campos, como qualquer outra classe. Geralmente usam-se os campos para anotar informações sobre o erro que provocou o lançamento da exceção, de forma a transmiti-las ao ponto de tratamento do erro, onde poderão ser usadas pela aplicação.

A seguir mostra-se uma aplicação que usa uma pilha e uma fila, cujas operações podem lançar os mesmos tipos de exceção, mas cada operação sempre anota no objeto de exceção lançado informações particulares, de tal forma que a aplicação, quando receber o objeto de exceção, possa identificar claramente a causa do erro. Inicialmente, declara-se uma classe de exceção para cada categoria de erro possível:

```
1  class Máximo extends Exception {
2      public byte id;
3      public Máximo(byte id) {this.id = id;}
4  }
5  class Mínimo extends Exception {
6      public byte id;
7      public Mínimo(byte id) {this.id = id;}
8  }
9  class Tamanho extends Exception {
10     public byte id;
11     public Tamanho(byte id) {this.id = id;}
12 }
```

Nas classes **Máximo**, **Mínimo** e **Tamanho**, o atributo público **id** serve para anotar informações sobre o ponto em que a falha for detectada. Usa-se o valor **1** para informar quando o erro for na pilha, e **2**, quando for na fila.

Observe, que, na classe abaixo, as exceções **Tamanho**, **Máximo** e **Mínimo** são lançadas, sendo registrado nos objetos de exceção o código do erro, no caso o inteiro **1**, para indicar que a exceção ocorreu na classe **PilhaDeInteiros**.

De propósito, a classe **PilhaDeInteiros** prevê o levantamento de exceção dentro de sua construtora, mas isso geralmente não é uma boa prática de programação porque frequentemente a expressão do tipo **new PilhaDeInteiros(...)** ocorre em declarações de campos. As exceções **Máximo** e **Mínimo** podem ser lançadas pelas operações **empilhe** e **desempilhe**, respectivamente.

```
1  public class PilhaDeInteiros {
2     private int[] item; private int topo = -1;
3     public PilhaDeInteiros(int n) throws Tamanho {
4        if (n < 0) throw new Tamanho(1);
5        item = new int[n];
6     }
7     public boolean vazia() {return (topo == -1);}
8     public int empilhe() throws Máximo {
9        if(topo+1 == item.length)throw new Máximo(1);
10       return item[++topo];
11    }
12    public int desempilhe() throws Mínimo {
13       if(vazia()) throw new Mínimo(1);
14       return item[topo--];
15    }
16 }
```

As exceções **Tamanho**, **Máximo** e **Mínimo** são também lançadas pelas operações de **FilaDeInteiros** abaixo, mas agora o código

de identificação da origem dos lançamentos é 2.

```
1  public class FilaDeInteiros {
2     private int inicio; private int fim;
3     private int tamanho;  private int[] item;
4     public FilaDeInteiros(int n) throws Tamanho {
5        if (n < 0) throw new Tamanho(2);
6        item = new int[n]; tamanho = n;
7     }
8     public boolean vazia() {return (inicio==fim);}
9     public void insira(int v) throws Máximo {
10       int p = (++fim) % tamanho;
11       if (p == inicio) throw new Máximo(2);
12       fim = p; item[fim]= v;
13    }
14    public int remova() throws Mínimo {
15       if(vazia()) throw new Mínimo(2);
16       frente = (++inicio % tamanho);
17       return item[inicio];
18    }
19 }
```

O programa **Aplicação** apresentado a seguir cria uma pilha **s** e uma fila **q** e realiza sobre essas estruturas diversas operações de inserir e remover itens, podendo ocorrer levantamento de exceções, tanto nas operações da pilha como da fila.

Observe que o fato de as construtoras de **PilhasDeInteiros** e **FilaDeInteiros** poderem levantar a exceção **Tamanho**, a cláusula **trows** do **main** de **Aplicação** deve propagá-la.

Suponha que o código não deixe claro a origem das exceções, que devem ser devidamente identificadas, para que se tome as providências cabíveis nas cláusulas **catch** correspondentes, como ilustra o programa **Aplicação**. Nesse caso, a inspeção do objeto de exceção lançado informa a origem do lançamento, permitindo a devida tomada de decisões em relação ao tipo de falha.

```
1  public class Aplicação {
2    public static void main(String args)throws Tamanho {
3      PilhaDeInteiros s = new PilhaDeInteiros(5);
4      FilaDeInteiros  q = new FilaDeInteiros(15);
5      for (int i=0; i<= 20; i++) {
6        try{... s.empilhe(i);... k = s.desempilhe();...
7            ... q.insira(i); ... k = q.remova();   ...
8        }
9       catch(Máximo e) {
10        if (e.id == 1) System.out.println("Foi na Pilha");
11        else System.out.println("Foi na Fila");
12      }
13      catch (Mínimo e){
14         System.out.println("Foi na Pilha ou na Fila");
15      }
16     }
17   }
18 }
```

Note que o teste da linha 10 da aplicação acima informa se a exceção capturada foi lançada pela pilha ou se foi pela fila.

7.5 Hierarquia de exceções

Pode-se criar uma hierarquia de classes de exceção para flexibilizar seu tratamento. A hierarquia permite agrupar o tratamento de erros quando isso for conveniente. Por exemplo, a hierarquia `Aritmético`, definida a seguir, agrupa as exceções `Máximo`, de estouro de valor máximo, `Mínimo`, de estouro de valor mínimo, e `Zero`, de divisão por zero.

```
class Aritmético extends Exception { }
class Máximo extends Aritmético { }
class Mínimo extends Aritmético { }
class Zero extends Aritmético { }
```

Pode-se escolher tratar todas de uma só forma, considerando-as do tipo `Aritmético`, conforme ilustra a classe `Usuário1` abaixo.

```
1  public class Usuário1 {
2     public static void main(String[] args) {
3        try {
4          ... throw new Máximo();
5          ... throw new Zero();
6          ... throw new Mínimo();
7          ... throw new E(); ...
8        }
9        catch(Aritmético  e) {
10          trata Aritmético e descendentes
11       }
12       catch(E e) { ... }
13       ...
14 }
```

Ou então tratar as exceções individualmente, como na classe `Usuário2` apresentada a seguir.

```
1  public class Usuário2 {
2    public static void main(String[] args) {
3      try {
4        ... throw new Máximo();
5        ... throw new Zero();
6        ... throw new Mínimo();
7        ... throw new E();
8        ...
9      }
10     catch(Máximo e) {trata Máximo e descendentes }
11     catch(Mínimo e) {trata Mínimo e descendentes }
12     catch(Zero e)   {trata Zero e descendentes }
13     catch(E e)      { ... }
14     ...
15 }
```

A hierarquização das exceções dá muita flexibilidade de programação, mas ressalta-se que a ordem de apresentação dos `catch`es não é livre. A ordem, de cima para baixo, deve ser de uma classe mais específica para a mais geral da hierarquia. Isso porque a busca pelo `catch` é feita de cima para baixo, sendo concluída quando for encontrado o primeiro que satisfizer o teste **é-um** relativo ao tipo do seu parâmetro.

A correção da ordem é automaticamente verificada pelo compilador, que, por exemplo, não aceita a ordenação de `catch`es usada no método `errado` do programa abaixo, porque o primeiro `catch` iria capturar todas as exceções de sua hierarquia, e, consequentemente, as cláusulas que o seguem seriam inúteis.

```
1  class A extends Exception { }
2  class B extends A { }
3  class OrdemErrada {
4     public void errado() {
5        try {... throw new B(); ...}
6        catch (Exception e ) { ... }
7        catch (A s) { ... }
8        catch (B s) { ... }
9     }
10 }
```

7.6 Informações agregadas

Objetos de classes que estendem `Throwable` ou `Exception` têm um campo contendo uma cadeia de caracteres que identifica o objeto de exceção.

Esse campo é usado como valor de retorno da operação `toString` associada ao objeto, e que é herdada da classe `Object`. Lembre-se que, em Java, se um objeto for usado onde uma cadeia de caracteres

é esperada, a operação `toString` é automaticamente chamada.

O uso dessa propriedade dos objetos de exceção é ilustrada pelo programa a seguir,

```
1  public class Agregado1 {
2    public static void main(String[] args) {
3       try {throw new ArrayStoreException();}
4       catch (ArrayStoreException e) {
5          System.out.println(e);
6       }
7       try {throw new ExceçãoDoUsuário();}
8       catch(ArrayStoreException e) {
9          String s = e.toString();
10         System.out.println(e + "\n" + s);
11      }
12   }
13 }
14
15 class ExceçãoDoUsuário extends ArrayStoreException { }
```

que imprime:

```
java.lang.ArrayStoreException
ExceçãoDoUsuário
ExceçãodoUsuário
```

Esse recurso é útil no processo de depuração do programa: a impressão do nome de uma exceção que não deveria ocorrer ajuda a localizar o erro.

Além disso, informações podem ser acrescentadas à descrição da exceção. Para isto basta incluir na construtora da classe de exceção uma chamada à construtora que tem um parâmetro do tipo `String` de sua superclasse `Exception` no momento da criação do objeto de exceção. Essa construtora acrescenta a cadeia passada via parâmetro ao descritor da classe de exceção.

O método `toString` do objeto de exceção retorna uma cadeia de caracteres formada pelo nome da classe do objeto concatenado

com `": "` (dois-pontos e um branco) e com a cadeia de caracteres passada à construtora de `Exception`.

Por exemplo, o programa abaixo imprime `Ex: Erro Previsto`.

```
1  class Ex extends ArrayStoreException {
2      Ex(String s) {super(s);}
3  }
4  public class Agregado2 {
5    public static void main(String[] args) {
6       try { throw new Ex("Erro Previsto");}
7       catch (ArrayStoreException e) {
8          System.out.println(e);
9       }
10   }
11 }
```

7.7 Conclusão

Tratamento de exceções é um recurso poderoso para tornar o código mais legível. A separaçao do código que realiza a função da aplicação do tratamento das situações que violam a consistência dos dados facilita a leitura e a manutenção do código.

Metodologicamente, recomenda-se que os códigos de tratamento de exceções, i.e., os corpos de `cathes`, sejam sucintos, realizando o absolutamente necessário e essencial para recuperarem-se do erro detectado, e, principalmente, não devem levantar exceções.

Exercícios

1. Por que exceções podem contribuir para aumentar o grau de reúso de uma classe?
2. Há alguma circunstância em que o uso de exceções é melhor que o de comandos condicionais?

3. O que são exceções verificadas?

4. Quando se deve usar exceções não-verificadas?

5. Como implementar a semântica de `finally` sem o uso dessa cláusula em um comando `try`.

Notas bibliográficas

Tratamento de exceções em Java é um conceito bem discutido em praticamente todos os textos sobre essa linguagem, e em particular no livro de K. Arnold, J. Gosling e D. Holmes [2, 3].

Capítulo 8

Biblioteca Básica

A biblioteca padrão `java.lang` provê classes que são fundamentais para o adequado uso da linguagem Java, e seu conhecimento é indispensável para o desenvolvimento de programas avançados.

As classes da biblioteca `java.lang` são automaticamente importadas para todo programa Java, podendo ser usadas livremente. Certamente, somente as classes usadas são incorporadas ao programa do usuário.

Essa biblioteca é bastante extensa e, em alguns pontos, seu domínio requer conhecimento mais avançado sobre o funcionamento da linguagem.

Entretanto, para facilitar o aprendizado da linguagem pelo leitor iniciante, apenas uma seleção de algumas das classes mais importantes dessa biblioteca é apresentada neste capítulo. E para cada classe selecionada também somente apresenta-se um subconjunto de suas operações.

Espera-se, desta forma, que o leitor iniciante na linguagem Java rapidamente capacite-se na escrita e compreensão de programas que contenham cálculos matemáticos, manipulem sequências de caracteres ou realizem conversão de tipos de dados, como transformação de sequências de caracteres numéricos em valores binários e o tratamento de tipos primitivos em um contexto de orientação por objetos.

As classes apresentadas neste capítulo estão reunidas em três importantes grupos:

- Cálculos matemáticos: `Math`
- Sequências de caracteres: `StringBuffer`, `StringTokenizer` e `String`.
- Classes-invólucro: `Integer`, `Boolean`, `Character`, `Byte`, `Long`, `Float` e `Double`.

8.1 Cálculos matemáticos

A biblioteca `java.lang` provê a classe `Math`, que define constantes e funções matemáticas. Essa classe possui métodos para realizar cálculos matemáticos comuns, como raiz quadrada, logaritmo e operações trigonométricas, conforme resume a Tabela 8.1. Seus membros são implementados como estáticos, podendo ser ativados sem a necessidade de se criar objetos da classe, devendo ser qualificados pelo nome da classe `Math`.

Os argumentos das funções matemáticas apresentadas na Tabela 8.1 são todos do tipo `double`, e os seus valores devem ser fornecidos em radianos.

As funções `round`, `ceil` e `floor` retornam um `int` e as demais retornam um valor do tipo `double`.

8.2 Cadeias de caracteres

A classe `String` implementa operações para manipulação de sequências de caracteres. Diferentemente de classes definidas pelo programador, `String` possui notação especial para suas constantes,

Função	Valor Retornado
`Math.PI`	π (aprox. 3.141592653589793)
`Math.E`	e neperiano (aprox. 2.718281828459045)
`Math.sin(a)`	seno de `a`
`Math.cos(a)`	co-seno de `a`
`Math.tan(a)`	tangente de `a`
`Math.asin(v)`	arco-seno de `v` $\in [-1.0, 1.0]$
`Math.acos(v)`	arco-seno de `v` $\in [-1.0, 1.0]$
`Math.atan(v)`	arcotangente(`v`) (ret. em $[-\pi/2, \pi/2]$)
`Math.atan2(x,y)`	arcotangente(`x/y`) (ret. em $[-\pi, \pi]$)
`Math.exp(x)`	e^x
`Math.pow(y,x)`	y^x
`Math.log(x)`	$\ln x$
`Math.sqrt(x)`	$\sqrt{x}$
`Math.ceil(x)`	menor inteiro $\geq$ `x`
`Math.floor(x)`	maior inteiro $\leq$ `x`
`Math.rint(x)`	inteiro mais próximo de `x`
`Math.round(x)`	`(int)floor(x + 0.5)`
`Math.abs(x)`	valor absoluto de `x`
`Math.max(x,y)`	máximo de `x` e `y`
`Math.min(x,y)`	mínimo de `x` e `y`

Tabela 8.1 Funções Matemáticas

também chamadas de literais, que são codificadas como sequências de caracteres do conjunto Unicode colocadas entre aspas (`"`).

Exemplos de literais do tipo `String` são:

`"" , " " , "\n" , "\r" , "\"" e "abcde"`,

onde, pela ordem, têm-se sequências de caracteres contendo, respectivamente, nenhum caractere, um caractere branco, o caractere *mudança de linha*, o caractere *retorno de cursor*, o caractere aspas (`"`) e a sequência de caracteres `abcde`.

Sequências de caracteres são objetos do tipo `String`, e o seu texto representa o nome de uma referência constante para o respectivo objeto e, portanto, pode ser atribuída a qualquer variável que seja do tipo referência para `String`. Por exemplo, a declaração e iniciação de uma referência `String s = "abcde"` cria a referência `s` e atribui a ela a referência ao objeto `String` alocado previamente para conter a sequência de caracteres `"abcde"`.

O operador binário `+` concatena duas sequências de caracteres. Seus operandos são referências para objetos do tipo `String`, a partir dos quais, constrói-se um terceiro objeto contendo a sequência formada pela justaposição das sequências representadas por seus operandos e retorna o endereço do objeto criado.

Expressões contendo somente literais `String` são avaliadas em tempo de compilação. Por exemplo, a expressão

```
"uma parte " + "outra parte"
```

gera exatamente o mesmo objeto que a sequência

```
"uma parte outra parte".
```

Diferentes ocorrências de um mesmo literal, mesmo em módulos compilados separadamente, geram um único objeto `String` para representá-los. Por outro lado, objetos `String` construídos durante a execução são sempre novos e distintos de qualquer outro objeto `String`, mesmo quando representam o mesmo valor.

A função `main` da classe `ConstanteString` abaixo cria três objetos do tipo `String`: dois durante compilação, `"xyz"` e `"abcde"`, que são alocados no início da execução do programa, e um outro durante a execução, na linha 5, como resultado da operação de concatenação.

```
1 public class ConstanteString {
2    public static void main(String[] args) {
3       String s1;
4       s1 = "xyz";
5       System.out.print("abcde" + s1 + "xyz");
6    }
7 }
```

Quando a função `main` da linha 2 da classe acima tiver iniciada sua execução, os objetos mostrados na Figura 8.1 serão imediatamente alocados.

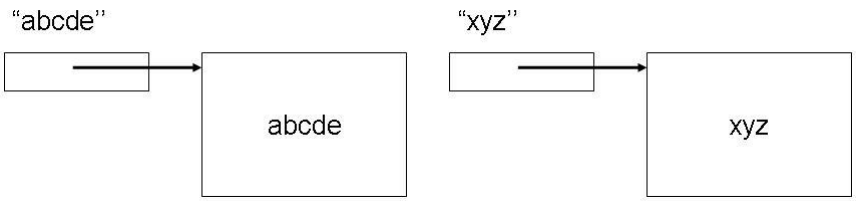


Figura 8.1 Constante String I

A declaração da linha 3 acrescenta a referência `s1`, como mostrado na Figura 8.2.

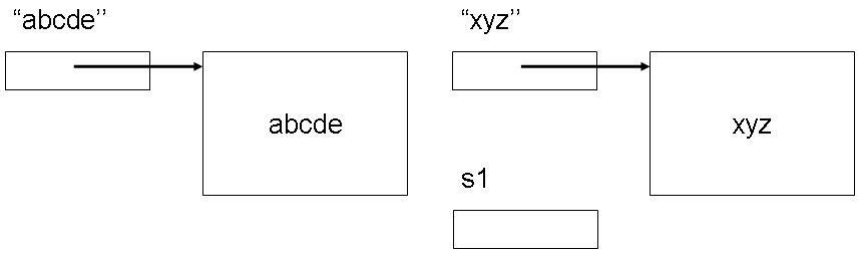


Figura 8.2 Constante String II

A atribuição da linha 4 cria o compartilhamento de objetos indicado na Figura 8.3.

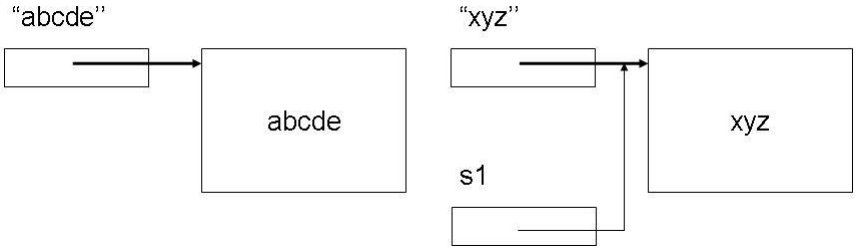


Figura 8.3 Constante String III

O valor impresso pelo comando na linha 5 é: `abcdexyzxyz`.

8.3 Classe `String`

A classe `String`[1] não possui operações para modificar o estado do objeto corrente. Por isto, objetos do tipo `String` são imutáveis

[1]Objetos do tipo String são frequentemente referidos como sequências de caracteres ou simplesmente sequências.

após sua criação, permanecendo constantes durante toda a execução. Isto dá ao programador a segurança de que o conteúdo de um campo `String` de um objeto qualquer nunca será alterado por terceiros, permitindo o compartilhamento seguro de sequências de caracteres entre diferentes objetos.

Nos casos em que se deseja trabalhar na estrutura de uma sequência de caracteres contidas em um objeto do tipo `String`, e.g., para alterá-la *in loco*, deve-se converter o objeto `String` em um objeto do tipo `StringBuffer`, que é suscetível à modificação de seu conteúdo, conforme descrito na Seção 8.4.

A classe `Object` possui uma operação definida com a assinatura `String toString()`. Assim, automaticamente, objetos de qualquer classe, tem, por herança, essa operação, que produz alguma informação textual sobre o objeto. Esse recurso é útil, por exemplo, no rastreamento da execução de um programa.

Para facilitar o tratamento de objetos como se fossem do tipo `String`, Java realiza conversão implícita, sempre que for requerido pelo contexto, de qualquer objeto para `String`, substituindo a ocorrência de uma referência `x` por `x.toString()`.

A operação `toString` default pode ser redefinida pelo programador. Por exemplo, o comando a linha 7 do programa

```
1  public class Entidade {
2    public String toString() {return "humano";}
3  }
4  public class A {
5    public static void main(String[] args) {
6       Entidade x = new Entidade();
7       String s = "Onde pode acolher-se um fraco " + x;
8       System.out.print(s);
9    }
10 }
```

é equivalente a

```
String s = "Onde pode acolher-se um fraco " + x.toString();
```

Dentre as várias operações definidas na classe **String**, somente algumas são discutidas aqui. O leitor interessado pode encontrar a lista completa das operações de **String** na página oficial de Java de endereço `http://www.oracle.com`.

As operações da classe **String** têm pelo menos um operando, que é a referência ao objeto receptor da operação, tornada disponível em todo método não-estático da classe pela referência implícita **this**. Por exemplo, a operação

```
s1.equals(s2)
```

compara a sequência de caracteres armazenada no objeto receptor apontado por **s1** com a sequência referenciada pelo argumento **s2**. Durante essa execução de **equals**, **this** tem o mesmo valor de **s1**. Na apresentação das operações de **String** a seguir, o termo *objeto receptor* refere-se ao objeto apontado por **this** no contexto da operação específica.

Arranjos de **String** podem ser iniciados diretamente em sua declaração, podendo-se omitir o operador **new** de alocação, como mostra o programa **ArranjoDeString1**, onde apenas são indicados os valores de iniciação do vetor. O arranjo de referências é automaticamente alocado e iniciado, como mostra a Figura 8.4, a qual retrata a situação após a execução dos comandos da linha 5.

```
1 public class ArranjosDeString1 {
2    public static void main (String[] args) {
3       String[] s = {"girafas", "tigres", "ursos"};
4       String[] r = new String[3];
5       r[0]="girafas"; r[1]="tigres"; r[2]="ursos";
6    }
7 }
```

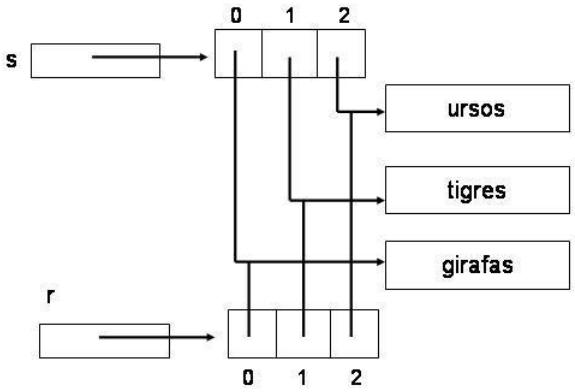


Figura 8.4 Arranjos de Strings

Note, na Figura 8.4, que os objetos contendo as cadeias de caracteres de mesmo valor são compartilhados. Isto é, embora haja duas ocorrências de "`girafas` ", "`tigres`" e "`ursos`", somente um objeto é alocado para as sequências repetidas.

A omissão do operador `new` na declaração acima é apenas uma conveniência de escrita. Pode-se, para alocação de arranjo iniciado na declaração, explicitamente especificá-lo, caso o programador assim o desejar, produzindo-se o mesmo efeito, conforme exemplificado o programa:

```
1 public class ArranjosDeString2 {
2    public static void main (String[] args) {
3       String[] s =
4            new String[]{"girafas","tigres","ursos"};
5       String[] r = new String[3];
6       r[0]="girafas"; r[1]="tigres"; r[2]="ursos";
7    }
8 }
```

Os programas `ArranjoDeString1` e `ArranjoDeString2` são equivalentes.

8.3.1 Construtoras de String

As funções construtoras de **String** mais usadas, dentre o total de 11 funções, são:

- **public String()**:
  contrói novo objeto **String** vazio.
  ```
  String s = new String();
  ```
- **public String(String t)**:
  constrói um novo objeto **String** contendo uma cópia da sequência de caracteres **t**.
  ```
  String s = new String("abcde");
  ```
- **public String(char v[])**:
  constrói um novo objeto **String** a partir de um vetor de **char**.
  ```
  char[] c = {'A', 'B', 'C', 'D'};
  String s  = new String(c);
  ```
- **public String(StringBuffer b)**:
  constrói um objeto **String** a partir de um objeto do tipo **StringBuffer** passado como parâmetro.

8.3.2 Tamanho de sequências

O tamanho da sequência de caracteres armazenada em um objeto **String** é dado pela operação:

- **public int length()**: retorna o número de caracteres do **String** apontado por **this**.
  ```
  String s = "abcde";
  int n = s.length();  // n recebe 5
  ```

8.3.3 Inspeção de caracteres

Pode-se ler um caractere de uma dada posição da sequência de caracteres armazenada em um objeto **String** pela operação:

- **public char charAt(int p)**:
  retorna o caractere na posição **p** da sequência armazenada no

objeto corrente. A posição `p = 0` denota a primeira posição na sequência, `p = 1`, a segunda, e assim por diante. O argumento `p` deve estar no intervalo [0, `this.length()-1`]. Valor de `p` fora desse intervalo causa `IndexOutOfBoundsException`. Exemplo de uso:

```
int i= 3;
String s= new String("abcdefgh");
char k = s.charAt(i); // k recebe o caractere 'd'
```

O programa **Contagem** abaixo determina o número de ocorrências de cada um dos caracteres ascii armazenados no objeto **String** referenciado por **s**. Para isso, o programa emprega um vetor de 256 posições, correspondentes ao conjunto ascii, para atuar como contador de ocorrências dos caracteres. Cada caractere recuperado da sequência dada somente é contado se estiver no intervalo [0, 255], de forma a indexar corretamente o vetor de contagem.

O teste da linha 9 é necessário, porque o conjunto de caracteres de Java é o Unicode, de 16 bits, e o conjunto ascii ocupa somente as suas 256 primeiras posições:

```
1  public class Contagem {
2     public static void main(String[] args) {
3        String s = "depois de tenebrosa tempestade, " +
4                                        "noturna sombra";
5        char c;
6        int [] contagem = new int[256];
7        for (int i=0; i < s.length(); i++) {
8            c = s.charAt(i);
9            if (c < 256) contagem[c]++;
10       }
11    }
12 }
```

8.3.4 Comparação de objetos String

Sequências de caracteres são representadas como objetos em Java e portanto variáveis do tipo `String` são referências para objeto desse tipo. Assim, quando se comparam objetos do tipo `String`, há duas possibilidades: comparam-se apenas as suas referências, ou então comparam-se os conteúdos das sequências contidas nos respectivos objetos. No primeiro caso, usa-se a operação `s1 == s2`, onde `s1` e `s2` são referências para objetos `String`, que apenas verifica se as referências `s1` e `s2` apontam para um mesmo objeto, isto é, se tratam do mesmo objeto.

Se se deseja comparar os conteúdos dos objetos apontados por `s1` e `s2`, deve-se usar uma das seguintes operações da classe `String`:

- `public boolean equals(Object x)`:
  retorna `true` se a sequência de caracteres do objeto apontado por `this` tiver o mesmo conteúdo que o do `String` representado pelo objeto apontado por `x`.
  ```
  boolean b = s1.equals(s2);
  ```
- `public boolean equalsIgnoreCase(Object x)`:
  retorna `true` se a sequência de caracteres do objeto apontado por `this` tiver o mesmo conteúdo que o do `String` representado pelo objeto `x`, considerando maiúsculo igual a minúsculo.
- `public int compareTo(String s)`:
  retorna um inteiro *menor que zero*, *zero* ou *maior que zero*, conforme a sequência de caracteres armazenada no objeto apontado por `this` seja lexicograficamente menor, igual ou maior que a do objeto apontado por `s`, respectivamente.
  ```
  int k = s1.compareTo(s2);
  ```

A operação `a.equals(b)` requer que `a` e `b` sejam referências para objetos do tipo `String` e verifica a igualdade das sequências de caracteres contidas nos objetos apontados por `a` e `b`. Obvia-

mente, `a` não pode ser `null`, mas `b` pode ser uma referência nula. Nesse caso, a função `equals` retorna sempre `false`, para qualquer que seja o objeto apontado por `a`. O programa abaixo imprime `Funcionou e são diferentes`.

```
1  public class SequênciaReferência {
2    public static void main(String[] args) {
3        String x = null;
4        if( "abc".equals("abc") )
5             System.out.print("Funcionou ");
6        else System.out.print("Não Funcionou ");
7        if( "abc".equals(x) )
8             System.out.print("e são iguais ");
9        else System.out.print("e são diferentes");
10   }
11 }
```

O uso da referência `"abc"`, no exemplo acima, como objeto receptor de uma operação da classe `String`, dá a certeza que a referência ao receptor não é nula, haja vista que todo literal denotando sequência de caracteres é devidamente alocado pelo compilador. Essa vantagem é destacada no próximo programa, que tem um erro na linha 7 somente detectável durante a execução.

```
1  public class ReferênciaNula {
2    public static void main(String[] args) {
3        String x = null;
4        if( "abc".equals(x) )
5             System.out.print("Iguais ");
6        else System.out.print("Diferentes");
7        if( x.equals("abc") )
8             System.out.print("Iguais ");
9        else System.out.print("Diferentes");
10   }
11 }
```

A construção da linha 4 é superior à da linha 7, porque a execução desta linha causa o aborto do programa, enquanto que a da linha 4 apenas informa que as sequências representadas por `x` e `"abc"` são diferentes. A implementação de `TabelaBinária` a seguir ilustra o uso da operação `compareTo`.

```
1  public class TabelaBinária {
2     private final static int MAX = 100;
3     private int tamanho;
4     private String[] tabela = new String[MAX];
5     public int inclui(String chave) throws Estouro {
6        int c, k = pesquisa(chave);
7        if (k != -1) return k;
8        if (tamanho == MAX ) throw new Estouro();
9        int novo = tamanho++;
10       for (int i = tamanho-1; i >= 1; i--) {
11           c = chave.compareTo(tabela[i-1]);
12           if (c < 0) tabela[i] = tabela[i-1];
13           else {novo = i ; break;}
14       }
15       tabela[novo] = chave;
16       return novo;
17    }
18    public int pesquise(String chave) {
19       int meio, c, inf = 0, sup = tamanho - 1;
20       while (inf <= sup) {
21          int meio = inf + (sup - inf)/2;
22          int c = chave.compareTo(tabela[meio]);
23          if (c == 0) return meio;
24          else if (c < 0) sup = meio - 1;
25               else inf = meio + 1;
26       }
27       return -1;
28    }
29 }
```

Nessa implementação, a tabela de chaves é mantida sempre or-

denada pelos valores crescentes da chave, de forma a possibilitar o uso da técnica de pesquisa binária. A função `t.pesquise(c)` retorna a posição da chave `c` na tabela `t`, caso ela esteja na tabela, do contrário retorna `-1`.

A inserção de uma chave inicia-se por sua pesquisa na tabela. Caso ela seja encontrada, apenas retorna-se o índice da chave na tabela. Caso contrário, a chave é inserida na tabela, respeitada a ordem lexográfica, provocando deslocamento dos elementos maiores do que ela.

A pesquisa por uma chave inicia-se com o espaço de busca igual a toda a tabela, e a cada iteração divide-se o espaço à metade, continuando a pesquisa pela chave na metade da tabela que ainda pode contê-la.

O comando da linha 21 de `TabelaBinária`, que calcula a posição do meio da porção da tabela sendo pesquisada, é equivalente a `(inf + sup)/2`, exceto que essa fórmula está sujeita a *overflow*, quando `sup` tiver valores próximos ao do maior `int` permitido na linguagem, por isso foi evitada.

8.3.5 Área de objetos canônicos

Comparar somente os valores de referências para objetos `String` é mais eficiente que comparar os conteúdos armazenados nos objetos por elas apontados, mas a escolha entre essas duas possibilidades é determinada pela semântica de cada aplicação.

Normalmente, é a comparação de conteúdo que se deve fazer. Entretanto, é possível forçar objetos `String` que representam a mesma sequência de caracteres a terem o mesmo endereço. Para isso, a classe `String` mantém internamente uma área de objetos canônicos do tipo `String`, que pode ser administrada pelo progamador com o auxílio do método `public String intern()`.

Quando uma chamada à `s.intern()` for executada, sendo `s` uma referência para objetos do tipo `String`, e a área de objetos canônicos privativos dessa classe já possuir um objeto contendo exatamente a mesma sequência de caracteres representada por `s`, segundo a operação `equals`, o endereço desse objeto da área privativa é retornado. Caso contrário, uma cópia do objeto `s` é instalada na área de objetos canônicos e retorna-se a referência dessa cópia. Assim, `s.intern()==t.intern()`, para duas referências quaisquer `s` e `t`, se e somente se `s.equals(t)`.

Todos os objetos `String` literais são automaticamente armazenados na área de objetos canônicos.

Considere o programa `Internos`:

```
1  public class Internos {
2    public static void main(String[] args) {
3      String abcde = "abcde", de = "de";
4      System.out.print((abcde == "abcde") + " ");
5      System.out.print((Local.abcde==abcde) + " ");
6      System.out.print((abcde==("abc" + "de")) + " ");
7      System.out.print((abcde==("abc" + de)) + " ");
8      System.out.println(abcde==("abc" + de).intern());
9     }
10 }
11 class Local {static String abcde = "abcde";}
```

A execução da função `main` acima produz a saída:

```
true true true false true
```

a qual mostra que `intern()` retorna o mesmo objeto canônico para as ocorrências do objeto literal `"abcde"`, para os objetos criados pelas operações `"abc" + "de"` da linha 6 e para a chamada `("abc" + de).intern()` da linha 8.

8.3.6 Localização de caracteres

Os caracteres que ocorrem dentro da sequência de caracteres armazenada em um objeto `String` são indexados a partir de `0`. Todas as operações para localizar a ocorrência de um caractere retornam a posição encontrada na sequência apontada por `this` ou então retorna `-1`, se o caracter `c` não for localizado.

As operações de localização de caracteres definidas pela classe `String` são:

- `public int indexOf(char c)`:
  retorna a posição (≥ 0) da primeira ocorrência do caractere `c` na sequência do `this`.
  ```
  String s = "bbbbabbbbbabbbbbabbbbabbbbax';
  int k = s.indexOf('a'); // k recebe valor 4
  ```
- `int indexOf(char c, int início)`:
  retorna a posição $\geq$ `início` da primeira ocorrência do caractere `c` na sequência do `this`.
  ```
  String s = "bbbbabbbbbabbbbbabbbbabbbbax';
  int k = s.indexOf('a',11); // k recebe 16
  ```
- `public int lastIndexOf(char c)`:
  retorna a posição da última ocorrência de `c` na sequência do `this`.
  ```
  String s = "bbbbabbbbbabbbbbabbbbabbbbax';
  int k = s. lastIndexOf('a'); // k recebe 26
  ```
- `public int lastIndexOf(char c, int início)`:
  retorna a posição $\leq$ `início` da última ocorrência de `c` na sequência do `this`.
  ```
  String s = "bbbbabbbbbabbbbbabbbbabbbbax';
  int k = s.lastIndexOf('a',11); // k recebe 10
  ```

Para ilustrar o uso das operações acima, considere a questão de determinar o número de caracteres entre a primeira e a última ocorrência de um mesmo caractere em uma dada sequência.

O método `numCaracteresEntre(s,c)` da classe `Localiza` abaixo retorna o número de caracteres localizados entre a primeira e a última ocorrência de um dado caractere `c` em um `String s`, passados com parâmetros. Caso o caractere em `c` não esteja duas vezes na sequência dada, essa função retorna o valor `-1`:

```
1  public class Localiza {
2     static int numCaracteresEntre(String s, char c) {
3        int inicio = s.indexOf(c);
4        if (inicio < 0) return -1;
5        int fim = s.lastIndexOf(c);
6        return fim - inicio - 1;
7     }
8     public static void main(String[] args) {
9        String s = ".Nas ondas vela pôs em seco lenho.";
10       System.out.print(numCaracteresEntre(s,"."));
11    }
12 }
```

O programa imprime `31`.

8.3.7 Localização de subsequências

As operações para localização de subsequências dentro de sequências de caracteres contidas em objetos do tipo `String` são:

- `public int indexOf(String s)`:
  retorna posição da primeira ocorrência, no objeto corrente, da sequência de caracteres armazenada em `s`.

  ```
  String s1 = "abc-dfg-hjk-dfg-gmn-opq-dfg-mhj-klm";
  String s2 = new String("dfg");
  int k = s1.indexOf(s2); // k recebe 4
  ```

- `public int indexOf(String s, int p)`:
  retorna posição $\geq$ `p` da primeira ocorrência de `s` na sequência do objeto `this`.

```
String s1 = "abc-dfg-hjk-dfg-gmn-opq-dfg-mhj-klm";
String s2 = new String("dfg");
int k = s1.indexOf(s2, 5); // k recebe 12
```

- `public int lastIndexOf(String s)`:
  retorna posição da última ocorrência do string `str` na cadeia armazenada em `this`.

```
String s1 = "abc-dfg-hjk-dfg-gmn-opq-dfg-mhj-klm";
String s2 = new String("dfg");
int k = s1.lastIndexOf(s2); // k recebe 24
```

- `public int lastIndexOf(String s, int p)`:
  retorna posição $\leq$ `p` da última ocorrência de `s` em `this`. O valor de `k` no fim do código abaixo é 12:

```
String s1 = "abc-dfg-hjk-dfg-gmn-opq-dfg-mhj-klm";
String s2 = new String("dfg");
int k = s1.indexOf(s2, 15); // k recebe 12
```

8.3.8 Particionamento de sequências

A operação `split` da classe `String` particiona a sequência de caracteres dada pelo objeto corrente em subsequências, com base em um conjunto de delimitadores. Essa operação devolve um arranjo de `String`, contendo os endereços dos objetos com as subsequências encontradas. A operação tem a seguinte assinatura:

```
public String[] split(String expressãoRegular)
```

onde o parâmetro `expressãoRegular` contém uma sequência de caracteres com a estrutura de uma expressão regular que descreva o conjunto de caracteres delimitadores das subsequências a serem identificadas.

Uma expressão regular em Java é uma cadeia de caracters que denota um padrão de ocorrência de caracteres, o qual consiste em uma sequência de caracteres, possivelmente entremeada de alguns símbolos especiais para especificar classes de caracteres. No caso

mais simples, a expressão é apenas uma sequência de caracteres, de comprimento maior que 0, que descreve o delimitador. Por exemplo, o padrão ":=" indica que essa sequência é o delimitador das subsequências.

Entretanto, padrões mais gerais podem ser definidos. É comum expressões regulares que denotam um conjunto de delimitadores, listados entre colchetes []. Por exemplo, o padrão "[:=*]" especifica que o delimitador de subsequências é um dos caracteres :, = ou *. E em "[a-z]", o delimitador é um caracter de a até z.

Pode haver padrões que são combinações de subpadrões. Alguns dos padrões permitidos estão exemplificados na Tabela 8.2.

Os programas Particiona1, Particiona2, Particiona3 e Particiona4, apresentados a seguir, imprimem as subsequências computadas e, para destacar o funcionamento do método split, usam o caractere - para separá-las na impressão dos resultados.

O programa Particiona1 abaixo mostra como é feita a partição de uma sequência usando o método split com base no delimitador formado pelo único caractere ':'.

```
1 public class Particiona1 {
2   public static void main(String[] args) {
3    String s   = "acc:xyz:dcc";
4    String[] t = s.split(":");// t = {"acc","xyz","dcc"}
5    for (String v : t) System.out.print(v + "-");
6    System.out.println("Tamanho de t = " + t.length);
7   }
8 }
```

A saída desse programa é:

```
acc-xyz-dccz-Tamanho de t = 3.
```

Observe que o delimitador ':' não é incluído nas subsequências retornadas. Nesse exemplo, a primeira partição vai do início da sequência até o caractere que antecede a primeira ocorrência do

Sumário dos Padrões de Expressão Regular	
Padrão	sequências que Casam com Padrão
`x`	caractere `x`
`\t`	caractere (`'\u0009'`)
`\n`	caractere *newline* (`'\u000A'`)
`\r`	caractere retorno de carro (`'\u000D'`)
`\f`	caractere *form-feed* (`'\u000C'`)
`\a`	caractere alerta (`'\u0007'`)
`\e`	caractere de escape (`'\u001B'`)
`[abc]`	um dos caracteres indicados: a, b, ou c
`[^abc]`	qualquer caractere exceto a, b, or c
`[a-zA-Z]`	caractere de a a z ou A a Z
`[a-d[m-p]]`	caractere de a a d, or m a p: [a-dm-p]
`[a-z&&[def]]`	caractere de d, e, or f (interseção)
`[a-z&&[^bc]]`	caractere de a a z, exceto b e c: [ad-z]
`[a-z&&[^m-p]]`	caractere de a a z, excluindo de m a p: [a-lq-z]

Tabela 8.2 Padrões de expressões regulares

delimitador '`:`'. A segunda subsequência vai do caractere que segue o primeiro '`:`' até o que antecede a segunda ocorrência do mesmo delimitador. A terceira é a subsequência que inclui os caracteres desde o `d` até o fim de `s`.

Considere agora o programa `Particiona2`, onde delimitadores justapostos forçam a criação de subsequências vazias:

```
1 public class Particiona2 {
2   public static void main(String[] args) {
3     String s   = "acc:xyz:dccz";
4     String[] u = s.split("c");
5                  // u = {"a", "" , ":xyz:d", "", "z"}
6     for (String v : u) System.out.print(v + "-");
7     System.out.println("Tamanho de u = " + u.length);
8   }
9 }
```

A saída do programa é:

```
a--:xyz:d--z-Tamanho de u = 5
```

Observe que a impressão de "--" indica que duas subsequências vazias foram encontradas.

O programa **Particiona3** destaca o fato de a operação **split** não incluir uma subsequência vazia quando o delimitador for o último caractere da sequência a ser particionada:

```
1 public class Particiona3 {
2   public static void main(String[] args) {
3     String s   = "acc:xyz:dccz";
4     String[] w = s.split("z"); // u = {"acc:xy" , ":dcc"}
5     for (String v : w) System.out.print(v + "-");
6     System.out.println("Tamanho de u = " + w.length);
7   }
8 }
```

A saída desse programa é:

```
acc:xy-:dcc-Tamanho de u = 2
```

O programa **Particiona4** abaixo ilustra a situação em que mais de um delimitador é usado para achar as partições. Especificamente, o programa a seguir separa as subsequências considerando como delimitador o caractere '+' ou o caractere ',', usados para formar a expressão regular da operação:

```
1 public class Particiona4 {
2   public static void main(String[] args) {
3     String a = "abc+cde,efg+hij";
4     String[] s;
5     s = a.split("[+,]"); // s = {"abc","cde","efg","hij"}
6     for (String x:a) System.out.print(x + "-");
7   }
8 }
```

Nesse caso, a saída é : **abc-cde-efg-hij**

Expressões regulares são também usadas em outras classes de Java como **Scanner** e **Pattern**, que são vinculadas a obtenção de valores de tipos primitivos a partir de um texto dado.

8.3.9 Construção de sequências

As seguintes operações são úteis para a construção de novas sequências de caracteres a partir de sequências dadas:

- `public String concat(String s)`:
  retorna uma nova sequência pela concatenação de uma cópia da sequência do objeto corrente com a de `s`.

```
String s = "tanta ";
String t = s.concat("tormenta");//t = "tanta tormenta"
String u = "a".concat("b").concat("c") // u = "abc"
```

- `public String replace(char velho, char novo)`:
  retorna uma sequência de caracteres idêntica à sequência em `this`, exceto que substituem-se todas as ocorrências do caractere dado por `velho` na sequência do objeto corrente pelo caractere `novo`. Se o caractere `velho` não ocorrer em `this`, o próprio valor da referência `this` é retornado. Caso contrário, um novo objeto `String` com as substituições comandadas é retornado.

```
String s = "papo".replace('p','t'); // s = 'tato';
```

- `public String substring(int início,int fim)`:
  retorna um novo objeto `String` que representa uma subsequência da sequência em `this`, que vai do caractere de posição `início` até a posição `fim`. A primeira posição tem índice 0.

```
String s = "A linguagem java".substring(12,15);
                // s = "java"
```

8.4 Classe `StringBuffer`

O conteúdo de um objeto do tipo `String` não pode ser alterado. Não há na classe `String` operação que permita alterar o conteúdo do objeto receptor, que, uma vez criado, retém a mesma sequência

de caracteres até ser descartado pelo programa. Essa restrição é útil para garantir que objetos do tipo `String` compartilhados não serão alterados.

Para implementar objetos contendo sequências de caracteres que possam ser alteradas *in loco*, deve-se usar a classe `StringBuffer` em vez de `String`.

Objetos do tipo `StringBuffer` armazenam internamente uma sequência de caracteres, chamada de **sequência buferizada**, cujo comprimento e conteúdo podem ser mudados via chamadas de certos métodos definidos nessa classe.

A operação `String x = "aa" + "bb"` é compilada para um código equivalente a

```
String x = new StringBuffer("aa").append("bb").toString();
```

que cria inicialmente uma sequência buferizada contendo `"aa"`, à qual acrescenta-se a sequência `"bb"`, após seu último caractere, pela operação `append`, expandindo, se necessário, a sequência original para acomodar a sequência `"bb"`. O objeto resultante, que é do tipo `StringBuffer`, é convertido para `String` via a operação `toString`.

As principais operações da classe `StringBuffer` são os métodos `append` e `insert`, para adicionar caracteres no fim de uma sequência ou em pontos específicos, respectivamente. Por exemplo, se

```
s = new StringBuffer("abcdefg"),
```

a chamada `s.append("xy")` atualiza a sequência em `s` para

`"abcdedfxy"`.

Note que `s.insert(4,"xy")` teria atualizado a mesma sequência para `"abcdxyefg"`.

Todo objeto `StringBuffer` é criado com uma capacidade máxima pré-definida para representar sua sequência buferizada. Caso essa capacidade do objeto torne-se insuficiente, a área interna da sequência é automaticamente aumentada.

8.4.1 Construtoras de `StringBuffer`

São três as principais funções construtoras de **`StringBuffer`**:

- `public StringBuffer()`:
  constrói um objeto com uma sequência de caracteres buferizada vazia com capacidade inicial de 16 caracteres
- `StringBuffer(int t)`:
  constrói um objeto com uma sequência buferizada vazia com capacidade inicial de `t` caracteres
- `StringBuffer(String s)`:
  constrói um objeto com uma sequência buferizada iniciada com uma cópia da sequência em `s`.

8.4.2 Inclusão de caracteres

Um subconjunto importante das operações de inclusão de caracteres na sequência buferizada de um objeto **`StringBuffer`** é formado pelos métodos:

- `public StringBuffer append(char c)`:
  acrescenta o caractere em `c` imediatamente após o último caractere da sequência do objeto `this`.
- `StringBuffer append(char[] v)`:
  acrescenta os caracteres do vetor `v`, pela ordem dos índices, após o último de `this`.
- `StringBuffer append(String s)`:
  acrescenta a sequência do objeto `s` no fim da sequência buferizada de `this`.
- `StringBuffer append(double d)`:
  acrescenta a sequência representativa do valor de `d` no fim da sequência `this`.

- `StringBuffer append(float f)`:
  acrescenta a sequência representativa do valor de `f` no fim da sequência `this`.
- `StringBuffer append(int i)`:
  acrescenta a sequência representativa do valor de `i` no fim da sequência `this`.
- `StringBuffer append(long g)`:
  acrescenta a sequência representativa do valor de `g` no fim da sequência `this`.
- `StringBuffer insert(int p, char c)`:
  insere o caractere em `c` na posição `p` da sequência buferizada de `this`, deslocando para a direita o conteúdo original uma posição a partir de `p`. Lança-se a exceção
  `IndexOutOfBoundsException`,
  se `p` não estiver no intervalo [`0`, `this.length()-1`].
- `StringBuffer insert(int p, char[] v)`:
  insere o conteúdo de `v` na sequência buferizada de `this` a partir de posição `p`, deslocando os caracteres da sequência original as posições necessárias para acomodar o conteúdo de `v`. Levanta-se a exceção
  `IndexOutOfBoundsException` ,
  se `p` não estiver no intervalo [`0`, `this.length()-1`].
- `StringBuffer insert(int p, String s)`:
  insere o conteúdo da sequência de caracteres do objeto `s` na sequência buferizada de `this` a partir de posição `p`, deslocando os caracteres da sequência original as posições necessárias. Levanta-se a exceção
  `IndexOutOfBoundsException` ,
  se `p` não estiver no intervalo [`0`, `this.length()-1`].

8.4.3 Controle de capacidade

As sequências buferizadas são criadas com um tamanho definido pelo usuário ou pelo valor default de 16. Esse tamanho é alterado automaticamente, se necessário, ou por ação explícita do programa.

As operações de `StringBuffer` relacionadas com o tamanho máximo da sequência buferizada são:

- `int capacity()`:
  retorna a capacidade da sequência buferizada do objeto corrente.
- `void ensureCapacity(int capacidadeMínima)`:
  toma as providências para que a sequência buferizada do objeto apontado por `this` tenha uma capacidade de no mínimo o valor especificado no parâmetro.
- `void setLength(int t)`:
  redefine a capacidade do `this` para o valor de `t`.
- `int length()`:
  retorna o número de caracteres na sequência em `this`.

8.4.4 Remoção de caracteres

As seguintes operações apagam um ou mais caracteres de uma sequência buferizada, compactando o resultado para não deixar espaços vazios entremeados:

- `StringBuffer delete(int início, int fim)`:
  remove os caracteres desde a posição `início` até `fim-1` da sequência buferizada do objeto corrente. Levanta-se a exceção
  `StringIndexOutOfBoundsException`
  se `início` estiver fora do intervalo [`0`, `this.length()-1`]. O método retorna o valor de `this`.

- `StringBuffer deleteCharAt(int p)`:
  remove o caractere da posição `p` da sequência buferizada, compactando-se o resultado. Se o parâmetro `p` estiver fora do intervalo [`0`, `this.length()-1`], lança-se o objeto de exceção
  `StringIndexOutOfBoundsException`.
  O método retorna o valor de `this`.
- `public String trim()`:
  remove espaços em branco e caracteres de controle ascii de ambas as extremidades da sequência `this`. Se a sequência `this` representa a sequência vazia, ou o primeiro e o último caractere dessa sequência tiverem valor maior que '\u0020' (caractere branco), a referência `this` é retornada. Se não houver caractere maior que '\u0020' na sequência `this`, uma nova sequência vazia é criada e retornada. Caso contrário, uma nova sequência, cópia da sequência `this`, mas com os caracteres de controle ascii e brancos iniciais e finais podados, é criada e retornada.

8.4.5 Recuperação de subsequências

As seguintes operações de `StringBuffer` servem para obter um ou mais dos caracteres armazenados em um objeto dessa classe nas posições dadas pelos seus parâmetros:

- `char charAt(int p)`:
  retorna o caractere da posição `p` da sequência buferizada de `this`. Levanta-se a exceção
  `IndexOutOfBoundsException` ,
  se `p` não estiver no intervalo [`0`, `this.length()-1`].
- `String substring(int início)`:
  retorna uma subsequência com a sequência de caracteres correntemente armazenados na sequência buferizada, desde a posi-

ção `início` até o seu fim. Levanta-se a exceção
`IndexOutOfBoundsException` ,
se `inic` não estiver no intervalo [0, `this.length()-1`].

- `String substring(int início, int fim):`
  retorna uma subsequência com os caracteres correntemente armazenados na sequência buferizada, desde a posição `início` até a posição `fim-1`. Lança-se a exceção
  `IndexOutOfBoundsException` ,
  se `início > fim` ou então `início` ou `fim` não estiverem no intervalo [0, `this.length()-1`].
- `String toString()`:
  retorna um objeto do tipo `String` contendo uma cópia da sequência buferizada do objeto corrente.
- `void getChars(int início,int fim,char[]d,int p)`:
  copia para as posições a partir de `p` do vetor `d` os caracteres contidos nas posições que vão de `início` a `fim-1` do `StringBuffer this`.
  Lança-se a exceção `IndexOutOfBoundsException` se
  (i)`início>fim`;
  (ii)`início` ou `fim` não estiverem em [0, `this.length()-1`] ou
  (ii) `p+fim-início > d.length()`.

8.4.6 Substituição de caracteres

As principais operações de `StringBuffer` para manipular os caracteres armazenados nos respectivos objetos, sem fazer cópias, são as seguintes:

- `StringBuffer replace(int inic,int fim,String s)`:
  substitui pelos caracteres em `s` os caracteres da subsequência que ocupa as posições de `inic` até `fim-1` ou até o fim da

sequência buferizada do objeto corrente, se `fim>=this.length()`. Em primeiro lugar, a porção especificada é removida, e somente depois é que o conteúdo de **s** é inserido. O valor **this** é retornado. Se o parâmetro **inic** for negativo, maior ou igual a **this.length()** ou maior que **fim**, lança-se a exceção
`StringIndexOutOfBoundsException`.

- **StringBuffer reverse()**:
  inverte a ordem dos caracteres da sequência de caracteres contida no objeto **this**. O valor de **this** é retornado.
- **void setCharAt(int index, char ch)**:
  muda para **ch** o valor do caractere da posição **index** de **this**.

8.4.7 Uso de StringBuffer

O programa **Substituição** troca todas as ocorrências do caractere **'a'** da sequência do objeto **assis** da classe **StringBuffer** por **'*'**:

```
1  public class Substituição {
2    public static void main(String[] args) {
3        StringBuffer assis = "Não se me daria olhar";
4        System.out.print( assis + " ==> ");
5        troca(assis,'a', '*');
6        System.out.println(assis);
7    }
8
9    public static void troca(StringBuffer s,char v,char n) {
10      for (int i = 0; i < s.length(); i++)
11           if (s.charAt(i) == v) s.setChar(i,n);
12   }
13 }
```

A saída é: `Não se me daria olhar ==> Não se me d*ri* olh*r`

O programa **DataDeHoje** apresentado abaixo mostra como acrescentar um texto no fim de uma sequência de caracteres.

A operação realizada pelo método **adiciona**, que converte a cadeia passada no primeiro parâmetro para um objeto **StringBuffer** e por meio da operação **insert** faz uma operação equivalente a uma concatenação, isto é, na linha 5, o programa realiza a operação equivalente a `t = s + " " + a + "."`.

A execução do método **ensureCapacity** da linha 13 dá a garantia de que, durante a execução da linha 14, a sequência buferizada referenciada por **c** têm o espaço necessário e não terá que ser expandida a cada nova inclusão, tornando o processo mais eficiente, pois cada expansão implica em criação de um novo objeto e realizar cópia da sequência de caracteres a ser expandida.

```
1  public class DatadeHoje {
2    public static void main(String[] args) {
3      String s = "Hoje é dia";
4      String a = "15 de outubro de 2015";
5      String t = adiciona(s,a); // t = s + a
6      System.out.println(t);
7    }
8
9    public static String adiciona(String b,String a) {
10     StringBuffer c = new StringBuffer(b);
11     int n1   = c.length();
12     int n2   = a.length();
13     c.ensureCapacity(n1 + n2 + 2);
14     c.insert(n1," ").insert(n1+1,a).insert(n1+n2+1,".");
15     return c.toString();
16    }
17 }
```

O programa produz a saída: `Hoje é dia 15 de outubro de 2015.`

O programa **Explicação** abaixo monta uma sequência de caracteres com a raiz quadrada de um número. A capacidade do objeto **StringBuffer** é estendida para acomodar uma sequência maior.

```
1  public class Explicação {
2     public static void main(String[] args) {
3        String s = raiz(256);
4        System.out.println(s);
5     }
6     public String raiz(int i) {
7        StringBuffer b = new StringBuffer();
8        b.append("A parte inteira de sqrt(").
9                                    append(i).append(')');
10       b.append(" = ").append(Math.ceil(Math.sqrt(i)));
11       return b.toString();
12    }
13 }
```

A saída é: `A parte inteira de sqrt(256) = 16`

8.5 Classe `StringTokenizer`

A classe **`StringTokenizer`** serve para particionar uma sequência de caracteres em *tokens*. Tipicamente um *token* é a maior subsequência de caracteres que não contém delimitadores.

Delimitadores são sequências de caracteres que participam das operações de obtenção dos tokens. Delimitadores, que podem ou não ser tratados como *tokens*, caso não sejam especificados, são pré-definidos, por default, pelo padrão `" \t\n\r\f"`, que os especifica como sendo: o caractere branco, o de tabulação, o de mudança de linha, o de retorno de carro e o de avanço de formulário. As funções construtoras da classe **`StringTokenizer`** são as seguintes:

- **`public StringTokenizer(String s)`**:
  constrói um objeto **`StringTokenizer`** para a sequência de caracteres **`s`**, usando os delimitadores *defaults*.
- **`StringTokenizer(String s, String d)`**:
  constrói um objeto **`StringTokenizer`** para a sequência de ca-

racteres `s`, usando como delimitadores de *tokens* os caracteres contidos na sequência `d`.

- `public StringTokenizer(String s, String d, boolean dToken)`:
  constrói o objeto `StringTokenizer` para a sequência `s`, usando delimitadores de *tokens* em `d`. O parâmetro `dToken` se igual a `true` indica que delimitadores devem ser tratados como *tokens*. Senão são apenas separadores. Se `d` for igual a `null`, a invocação de operações do objeto criado causa a exceção `NullPointerException`.

As funções construtoras de `StringTokenizer` sempre definem a posição corrente do próximo *token* para o início do primeiro *token* na sequência de caracteres do objeto corrente.

8.5.1 Operações de `Tokenizer`

As seguintes operações permitem a extração um a um dos *tokens* contidos na sequência de caracteres informada na criação do objeto `Tokenizer`:

- `public int countTokens()`:
  informa o número de vezes que o método `nextToken` pode ser chamado antes que ele lance uma exceção por falta de *tokens*. A posição corrente do próximo *token* não é alterada.
- `boolean hasMoreTokens()`:
  informa se ainda existem *tokens* disponíveis no objeto corrente a partir da posição corrente. Essa operação deve ser invocada antes de cada chamada a `nextToken`.
- `public String nextToken()`:
  retorna o próximo *token* reconhecido na sequência do objeto corrente e avança a posição corrente para após esse `token`.

Lança-se a exceção **NoSuchElementException** se não houver **token** a ser retornado.

- **public String nextToken(String d)**:
  troca o conjunto de delimitadores de *token* do objeto corrente para a sequência **d**, retorna o próximo *token* reconhecido na sequência do objeto corrente e avança a posição corrente para após o **token** reconhecido. Lança-se a exceção **NoSuchElementException** se não houver **token** a ser retornado. Após a chamada, o conjunto de delimitadores **d** torna-se o conjunto *default* para o objeto corrente.

Note que os delimitadores de *tokens* podem ser informados à função **nextToken**, que é responsável para recuperar, um a um, os tokens contidos em uma sequência. Se esses delimitadores forem omitidos na chamada de **nextToken**, valores *defaults* são usados. Os delimitadores podem também ser especificados no momento da criação do objeto **StringTokenizer**.

No exemplo a seguir, os delimitadores são os definidos por *default* e não são tokens. Desta forma, o efeito do programa abaixo é separar as palavras da sequência de caracteres dada, tomando espaço em branco como delimitador de *tokens*, e imprimir os *tokens* recuperados, separando-os por '-'.

```
1 public class Escandimento {
2   public static void main(String[] args) {
3     StringTokenizer t =
4        new StringTokenizer("Se acabe o nome e glória");
5     while (t.hasMoreTokens()) {
6          System.out.print(t.nextToken() + "-");
7     }
8   }
9 }
```

O programa imprime: **Se-acabe-o-nome-e-glória-**.

8.6 Classes Invólucro

As classes invólucro são usadas para dar status de objeto a valores de tipos primitivos. Elas promovem o empacotamento ou desempacotamento de valores de tipos primitivos, permitindo-se que esses valores sejam usados onde objetos são esperados.

Os tipos primitivos de Java são implementados com um modelo de armazenamento denominado **semântica de valor**, enquanto os demais tipos, introduzidos por meio de classes, operam com o modelo **semântica de referência**.

Na semântica de valor, os elementos declarados recebem diretamente valores do tipo de dado com o qual foram declarados. Por exemplo, quando declaram-se `int x = 10, y = 20`, os elementos declarados `x` e `y` armazenam inteiros, no caso, os valores `10` e `20`, respectivamente. A atribuição `x = y` copia o valor inteiro em y para a célula de memória alocada para `x`.

Por outro lado, na semântica de referência, os elementos declarados recebem o endereço de um objeto e não o próprio objeto. Assim, a declaração `Pessoa p = new Pessoa()` introduz `p` não como um objeto do tipo `Pessoa`, mas como uma referência para o objeto criado pelo operador `new`. Se `q` tiver sido declarado com o tipo `Pessoa`, a atribuição `q = p` não faz cópia do objeto associado a `p`, mas apenas copia o valor do endereço contido em `p`.

A adoção do modelo de semântica de valor para tipos primitivos tem a razão pragmática de oferecer maior eficiência no consumo de memória e em tempo de processamento na manipulação de tipos básicos. Por outro lado, o uso de um modelo único para todos os tipos de dados tornaria a programação mais uniforme e ofereceria maior grau de reusabilidade de componentes de software.

A solução de compromisso de Java é a definição, na biblioteca `java.lang`, de um conjunto de **classes invólucro**, uma para

cada um dos tipos primitivos da linguagem, a saber:

Tipo	Classe Invólucro	Tipo	Classe Invólucro
boolean	Boolean	char	Character
byte	Byte	short	Short
int	Integer	long	Long
float	Float	double	Double

Todas as classes invólucro incluem funções construtoras para encaixotar valores de tipos primitivos como objetos e operações para desencaixotá-los de volta. O programa `Invólucro` abaixo mostra como valores de tipos primitivos podem ser empacotados e desempacotados:

```
1  public class Invólucro {
2      int i, j = 50;
3      // empacotamento em k do inteiro j
4      Integer k = new Integer(j);
5      // empacotamento do inteiro em k
6      i = k.intValue();
7      float x, y = 100.0;
8      // empacotamento em z do float y
9      Float z = new Float(y);
10     // desempacotamento do float em z
11     x = z.FloatValue();
12     System.out.print
13         ("i= " +i+ " j= " +j + " x= " +x+ " y= " +y);
14 }
```

A saída do programa é : `i= 50 j= 50 x= 100.0 y= 100.0`

As classes invólucro possuem, além das operações de empacotamento e desempacotamento de valores de tipos primitivos, métodos estáticos de interesse geral para conversão de sequências de caracteres em valores numéricos, determinação de categoria de caracteres, conversão de letras maiúsculas em minúsculas e vice-versa, etc. Para conhecer a lista completa de operações dessas classes, sugere-se uma visita à página da Oracle.

8.6.1 Classe `Boolean`

Objetos do tipo `Boolean` encaixotam valores do tipo `boolean`. Suas operações principais são:

- `public static Boolean valueOf(boolean b)`
- `public static Boolean valueOf(String s)`
- `public static String toString(boolean b)`
- `public static boolean parseBoolean(String s)`

A função `toString` retorna a sequência `"TRUE"` ou `"FALSE"`, conforme seja o valor booleano `b` informado.

8.6.2 Classe `Character`

Objetos do tipo `Character` representam valores primitivos do tipo `char`. Principais operações são:

- `public String toString()`
- `public char charValue()`
- `public static boolean isLetterOrDigit(char c)`
- `public static boolean isLowerCase(char c)`
- `public static boolean isUpperCase(char c)`
- `public static boolean isSpace(char c)`
- `public static char toLowerCase(char c)`
- `public static char toUpperCase(char c)`
- `public static boolean isDigit(char c)`
- `public static int digit(char c, int radix)`
- `public static char forDigit(int digito,int radix)`

Nas operações `digit` e `forDigit`, o parâmetro `radix` é a base do sistema de numeração do dígito contido no outro parâmetro, normalmente base 10. O método `forDigit` devolve o caractere que representa o dígito, dada a base `radix`.

8.6.3 Classe `Number`

`Number` é uma classe abstrata estendida por `Integer`, `Long`, `Float` e `Double`, as quais encaixotam os respectivos tipos primitivos e definem métodos para fornecer o valor numérico empacotado pelo tipo correspondente, isto é, para `int`, `long`, `float` ou `double`.

- `public byte byteValue()`
- `public abstract double doubleValue()`
- `public abstract float floatValue()`
- `public abstract int intValue()`
- `public abstract long longValue()`
- `public abstract short shortValue()`

8.6.4 Classe `Integer`

Subclasse de `Number` com as operações:

- `public static Integer valueOf(String s)`
- `public static Integer valueOf(String s,int radix)`
- `public Integer intValue(int i)`
- `public static int parseInt(String s)`
  `throws NumberFormatException`
- `public String toString()`

O método `parseInt` converte uma sequência de caracteres contendo apenas algarismos decimais em um valor do tipo `int`. A presença de qualquer outro tipo de caractere na sequência causa o lançamento da exceção indicada.

8.6.5 Classe `Long`

Subclasse de `Number` com as operações:

- `public static Long valueOf(String s)`

- `public static Long valueOf(String s, radix r)`
- `public static Long valueOf(long n)`
- `public static long parseLong(String s)`
  `throws NumberFormatException`
- `public String toString()`

O método `parseLong` converte uma sequência de caracteres contendo apenas algarismos decimais em um valor do tipo `Long`. A presença de qualquer outro tipo de caractere na sequência causa o lançamento da exceção indicada.

8.6.6 Classe `Float`

Subclasse de `Number` com as operações:

- `public static Float valueOf(String s)`
- `public static Float valueOf(float f )`
- `public static float parseFloat(String f)`
- `public static String toString()`

8.6.7 Classe `Double`

Subclasse de `Number` com as operações:

- `public static Double valueOf(double d)`
- `public static Double valueOf(String s)`
- `public static Double parseDouble(String s)`
- `public String toString()`

8.7 Conclusão

Todo programador Java precisa conhecer em detalhes a biblioteca `java.lang`, que é automaticamente incorporada a todo programa. Classes como `Math`, `String`, `StringBuffer` e as chama-

das invólucro estão presentes em quase todas as aplicações. Desconhecer detalhes de como são operadas torna muito difícil ler ou escrever programas de alguma valia.

Exercícios

1. Quando deve-se usar a classe `String` e quando deve-se usar classe `StringBuffer`?
2. Qual é o tipo de um literal formado por uma sequência de caracteres?
3. Por que classes invólucro são necessárias?

Notas bibliográficas

A melhor referência bibliográfica para conhecer em profundidade a biblioteca `java.lang` é a página oficial da Oracle [43] para a linguagem Java.

Capítulo 9

Entrada e Saída

Arquivos que armazenam dados como uma sequência de *bytes* ou de caracteres são ditos arquivos de fluxos baseados em *bytes* ou baseados em caracteres, respectivamente.

A biblioteca `java.io` de Java para administar arquivos de fluxo de entrada e saída é rica e sofisticada, contendo cerca de 64 classes. Para um programador iniciante, o aprendizado dessa biblioteca pode ser difícil, sendo recomendado seu estudo após se adquirir um pouco mais de fluência na linguagem.

Entretanto, é indispensável que operações de entrada e saída possam ser incorporadas aos primeiros programas o mais cedo possível para torná-los mais atraentes. Assim, para simplificar o aprendizado, apenas um subconjunto essencial das classes que implementam entrada e saída de fluxo de bytes é apresentado neste capítulo, e dessas classes, somente as operações mais simples, mas que sejam suficientes para desenvolver aplicações interessantes.

9.1 Fluxos de *bytes* e caracteres

As classes do pacote `java.io` que implementam arquivos de fluxo de bytes e caracteres são: `InputStream`, `OutputStream`, `System`, `PrintStream`, `FilterOutputStream`, `FileInputStream`, `Reader`, `DataInputStream` e `BufferedReader`.

9.1.1 Classe `InputStream`

A classe `InputStream` é abstrata e possui operações definidas para ler arquivo de fluxo *byte* a *byte*, retornando, a cada operação de leitura, o valor inteiro lido no intervalo `[0,255]` ou então `-1` se o fim do fluxo de entrada tiver sido alcançado. Normalmente, essa classe não é usada diretamente pelo programador. Seu papel principal é servir como superclasse de todas as que implementam arquivos de entrada de tipo fluxo de *bytes*.

9.1.2 Classe `OutputStream`

A classe `OutputStream` é uma classe abstrata com operações para escrever o valor de um `byte`, i.e., um valor no intervalo `[0,255]`, no arquivo de saída de fluxo de *bytes*. Ela é usada como superclasse de todas as classes que representam um fluxo de saída de *bytes*.

9.1.3 Classe `FilterOutputStream`

Classe `FilterOutputStream` é uma extensão concreta da classe `OutputStream`. Sua construtora aceita como parâmetro objetos da hierarquia de `OutputStream`. Objetos `FilterOutputStream` agem como filtro para objetos do tipo `OutputStream`. Suas operações são redefinições das operações de `OutputStream` e realizam transformações nos dados antes de enviá-los ao arquivo de fluxo subjacente, que foi passado à construtora. Todas as classes que filtram fluxo de saída têm como superclasse `FilterOutputStream`.

9.1.4 Classe `PrintStream`

A classe `PrintStream` é subclasse de `FilterOutputStream` que provê métodos para enviar valores de tipos básicos e sequências

de caracteres para arquivo de fluxo de *bytes*. Suas principais operações são:

- `public PrintStream(OutputStream arquivo)`:
  cria objeto para imprimir tipos primitivos e sequências de caracteres no arquivo de fluxo de *bytes* passado como parâmetro.
- `public void print(char c)`:
  envia o caractere `c` para o arquivo de fluxo subjacente.
- `public void print(int i)`:
  envia o inteiro `i` para o arquivo de fluxo subjacente.
- `public void print(float f)`:
  envia o valor fracionário `f` para o arquivo de fluxo subjacente.
- `public void print(boolean b)`:
  envia o booleano `b`, convertido para sequência de caracteres (`TRUE` ou `FALSE`) para o arquivo de fluxo subjacente.
- `public void print(String s)`:
  envia a sequência de caracteres `s` para o arquivo de fluxo subjacente.

9.1.5 Classe `System`

A classe `System` dá acesso aos arquivos de fluxo `in`, `out` e `err`, automaticamente criados para leitura de teclado, escrita na tela do monitor e no arquivo de erro, respectivamente. Esses campos da classe `System` são definidos da seguinte forma:

- `public static InputStream in`
- `public static PrintStream out`
- `public static PrintStream err`

O uso direto arquivo `in` é limitado por ser de fluxo de *bytes*, tornando muito trabalhosa, por exemplo, a leitura de um inteiro digitado na entrada padrão. Cada *byte* do inteiro teria que ser lido separadamente, e o valor do inteiro devidamente construído. Em

geral, o arquivo `in` é usado para apenas designar a entrada padrão, sendo as operações realizadas por classes derivadas de nível mais elevado de abstração. Por outro lado, os objetos `out` e `err` são de uso corrente.

9.1.6 Interface `DataInput`

A interface `DataInput` define as assinaturas das várias operações, que devem ser implementadas segundo o contrato definido para cada uma. A classe `FilterInputStream`, que faz parte dessa biblioteca, é uma classe que implementa `DataInput`. As principais operações de `DataInput` e seus respectivos contratos são:

- `boolean readBoolean()`:
  lê de um *byte* e retorna `true` se for diferente de zero, `false`, caso contrário.
- `byte readByte()`:
  lê e retorna um *byte*.
- `char readChar()`:
  lê dois *bytes* e retorna um valor do tipo `char`.
- `double readDouble()`:
  lê oito *bytes* e retorna um valor do tipo `double`.
- `float readFloat()`:
  lê quatro *bytes* e retorna um valor do tipo `float`.
- `int readInt()`:
  lê quatro *bytes* e retorna um valor do tipo `int`.
- `String readLine()`:
  lê a próxima linha de texto do fluxo de entrada.
- `long readLong()`:
  lê oito *bytes* e retorna um valor do tipo `long`.
- `short readShort()`:
  lê dois *bytes* e retorna um valor do tipo `short`.

9.1.7 Classe `FileInputStream`

A classe `FileInputStream` é uma subclasse de `InputStream` para permitir a leitura de *bytes* de um arquivo específico. No momento da criação de um objeto dessa classe, pode-se passar como parâmetro para a construtora

```
FileInputStream(String nome)
```

uma sequência de caracteres que seja nome do caminho de um arquivo no sistema de arquivo. A construtora abre o arquivo especificado e estabelece para ele uma conexão via o objeto criado. Os comandos básicos de leitura de *bytes* herdados de `InputStream` são redirecionados para o arquivo aberto.

O uso direto de `FileInputStream` também é restrito. Sua função é prover uma estrutura básica para construir classes de entrada de dados mais elaboradas, como `DataInputStream`.

9.2 Fluxo de valores primitivos

A leitura e escrita de valores de tipos primitivos são implementadas pelas classes `DataInputStream`, `Reader`, `InputStreamReader` e `BufferedReader`, detalhadas a seguir.

9.2.1 Classe `DataInputStream`

`DataInputStream` é subclasse da classe `FilterInputStream` que implementa a interface `DataInput`. Essa classe permite a leitura de valores de dados primitivos de Java de um arquivo de fluxo de *bytes*, definido pela criação do objeto `DataInputString`. Pode-se gravar dados via objetos do tipo `DataOutputStream` e lê-los de volta por meio de um objeto `DataInputStream`. As principais operações dessa classe são:

- `public DataInputStream(InputStream arquivo)`:
  cria um `DataInputStream` que usa o arquivo de fluxo definido no parâmetro `arquivo`, que pode ser uma referência para um objeto do tipo `FileInputStream`.
- `boolean readBoolean()`:
  conforme `DataInput`.
- `byte readByte()` :
  conforme `DataInput`.
- `char readChar()` :
  conforme `DataInput`.
- `double readDouble()`:
  conforme `DataInput`.
- `float readFloat()`:
  conforme `DataInput`.
- `int readInt()` :
  conforme `DataInput`.
- `String readLine()`:
  conforme `DataInput`.
- `void close()`:
  fecha o arquivo de fluxo e libera os recursos associados ao fluxo.

O protocolo de uso de `DataInputStream` é ilustrado pelo código:

```
1 FileInputStream f = new FileInputStream("arquivo.dat");
2 DataInputStream g = new DataInputStream(f);
3 int k    = g.readInt();
4 double s = g.readDouble();
```

que destaca a necessidade de inicialmente criar, na linha 1, o objeto de acesso ao arquivo, no exemplo, o de nome `"arquivo.dat"`, e utilizá-lo na construtora de `DataInputStream` na linha 2.

Para mostrar o funcionamento da classe `DataInputStream` e do método `String.split`, muito usados para manipular dados

lidos de arquivo de fluxo, suponha que seja dado o arquivo de fluxo **meusdados.txt** contendo a seguinte sequência de caracteres:

```
"ab1c cd2ef,ghi jk3mnop,qrs4tuvz !@#$5%^&*() ZZZ\naaaaa"
```

onde **\n** é o caractere de mudança de linha.

O programa **Io1** a seguir faz a leitura de uma linha do arquivo de fluxo **meusdados.txt**, a particiona nas maiores subsequências que não contêm vírgula (**","**) e imprime essas subsequências separando-as por **"--"**:

```
1  public class Io1 {
2    public static void main(String[]args) throws Exception {
3      DataInputStream in;
4      in = new DataInputStream
5             (new FileInputStream("meusdados.txt"));
6      String s = in.readLine();  // deprecated
7      String[] a = s.split("[,]"); // separados por ,
8      for (String x : a)System.out.print(x + "--");
9      System.out.println();
10   }
11 }
```

Saída: **ab1c cd2ef--ghi jk3mnop--qrs4tuvz !@#$%^&*() ZZZ--**

9.2.2 Classe Reader

Reader é uma subclasse abstrata para leitura de sequências de caracteres em arquivo de fluxo de *bytes*, na forma definida por suas subclasses.

9.2.3 Classe InputStreamReader

Um objeto **InputStreamReader** faz a ponte de um fluxo de *bytes* para um fluxo de caracteres. Ele faz a leitura dos *bytes* do fluxo e os decodifica conforme um mapeamento entre caracteres unicode e

bytes, fornecido pela aplicação ou pela plataforma. Sua operação mais importante é:

- `public InputStreamReader(InputStream in)`:
  cria um `InputStreamReader` que usa o mapeamento *default* de unicode a *bytes*.

O uso dessa classe como suporte à entrada de dados realizada por meio da classe `BufferedReader` segue o seguinte protocolo:

```
BufferedReader in
  = new BufferedReader(new InputStreamReader(System.in));
```

no qual o teclado é tratado como uma entrada buferizada.

9.2.4 Classe `FileReader`

A classe `FileReader` implementa a leitura de texto decodificando os *bytes* lidos segundo um conjunto de caracteres (`Charset`) especificado ou por um conjunto *default* da plataforma.

Os conjuntos do tipo `java.nio.charset.Charset` definem mapeamentos entre sequências de caracteres unicode de 16-bits e sequências de *bytes*. Os conjuntos disponíveis em toda plataforma Java são `US-ASCII`, `ISO-8859-1`, `UTF-8`, `UTF-16BE`, `UTF-16LE` e `UTF-16`, dentre eles destacam-se:

- `US-ASCII`: bloco de caracteres latinos codificados como ascii.
- `ISO-8859-1`: alfabeto latino, incluindo letras acentuadas.

As principais operações da classe `FileReader` são as seguintes:

- `FileReader(File arquivo)`:
  cria arquivo de leitura de texto que usa caracteres do conjunto *default* da plataforma.
- `FileReader(FileDescriptor arquivo)`:
  cria arquivo de leitura de texto que usa caracteres do conjunto *default* da plataforma.

- `FileReader(String arquivo)`:
  cria arquivo de leitura de texto que usa caracteres do conjunto *default* da plataforma.
- `FileReader(File arquivo, Charset conjunto)`:
  cria arquivo de leitura de texto que usa caracteres do conjunto `Charset` dado.
- `FileReader(String arquivo, Charset conjunto)`:
  cria arquivo de leitura de texto que usa caracteres do conjunto `Charset` dado.

9.2.5 Classe `BufferedReader`

Objetos `BufferedReader` leem um texto a partir de um fluxo de caracteres e o armazena em um *buffer* interno, do qual porções são retiradas à medida que comandos de leitura de caracteres, de arranjos ou de linhas são emitidos. Os comandos de leitura causam pedidos de leitura correspondentes ao fluxo de dados subjacente.

`BufferedReader` opera sobre os fluxos básicos da família `Reader`, e.g., `InputStreamReader`. Suas principais operações são:

- `BufferedReader(Reader in)`:
  abre para leitura o fluxo de caracteres definido pelo parâmetro e usa um *buffer* de um tamanho *default*.
- `void close()`:
  fecha o arquivo de fluxo representado pelo objeto corrente.
- `int read()`:
  lê um caractere.
- `int read(char[ ] c, int p, int t)`:
  lê uma sequência de `t` caracteres e a armazena a partir da posição `p` do arranjo `c`.
- `String readLine()`:
  lê uma sequência de caracteres até encontrar o fim de linha.

O programa `TestaConsole` abaixo obtém informações do usuário, via o arquivo padrão de entrada `System.in`. Esse programa espera que o usuário digite um número inteiro seguido do pressionamento da tecla *ENTER*. O valor digitado é lido como uma sequência de caracteres, que é então convertida para o valor inteiro que representa e é processado adequadamente:

```
1  public class TestaConsole {
2    public static void main(String[] args)
3                                throws IOException {
4      Carteira carteira = new Carteira();
5      BufferedReader console = new BufferedReader(
6                   new InputStreamReader(System.in));
7      System.out.println("Quantas moedas de 5 centavos?");
8      String s = console.readLine();
9      carteira.inclua5(Integer.parseInt(s));
10     System.out.println("Quantas moedas de 10 centavos?");
11     s = console.readLine();
12     carteira.inclua10(Integer.parseInt(s));
13     System.out.println("Quantas moedas de 25 centavos?");
14     s = console.readLine();
15     carteira.inclua25(Integer.parseInt(s));
16     double total = carteira.getTotal();
17     System.out.println("O total é R$ " + total);
18     System.exit(0);
19   }
20 }
21 class Carteira {
22    public void inclua5(int n) { cinco += 0.05*n; }
23    public void inclua10(int n) { dez += 0.10*n; }
24    public void inclua25(int n) { vinteCinco += 0.25*n ; }
25    public double getTotal(){return(cinco+dez+vinteCinco);}
26 }
```

O programa `Io1` apresentado usa o método `readLine` que atualmente está depreciado. A classe `Io2` abaixo é uma versão atua-

lizada de `Io1`, que usa classe `BufferedReader` no lugar da classe `DataInputStream`.

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.nio.charset.Charset;
4  public class Io2 {
5    public static void main(String[]args) throws Exception {
6      Charset charset = Charset.forName("ISO-8859-1");
7      BufferedReader in = new BufferedReader(
8                  new FileReader("meusdados.txt", charset));
9      String s = in.readLine();
10     String[] a = s.split("[,]"); // separados por ,
11     for (String x : a) System.out.print(x + "--");
12     System.out.println();
13   }
14 }
```

9.3 Leitura de tipos primitivos

O pacote `java.util` disponibiliza a classe `Scanner` para facilitar a leitura de cadeias de caracteres e dela obter valores de tipos primitivos e subcadeias de caracteres.

A construtora de `Scanner` aceita como parâmetro um arquivo da hierarquia de `File`, como `System.in`, ou então uma cadeia de caracteres, a partir dos quais os valores são obtidos.

Suas principais operações são as seguintes:

- `Scanner(File fonte)`:
  constrói um *scanner* que produz valores pela leitura do arquivo `fonte`.
- `Scanner(String fonte)`:
  constrói um *scanner* que produz valores retirados da cadeia de caracteres `fonte`.

- `Scanner useDelimeter(String padrão)`:
  define o padrão, que é uma expressão regular definida na classe `Pattern` de Java, e que delimita os elementos a ser obtidos da entrada dada.
- `boolean hasNext()`:
  retorna `true` se o próximo *token* casa com o padrão definido.
- `boolean hasNextT()`, onde T deve ser substituído por `Int`, `Byte`, `Sort`, `Long`, `Double`, `Float` ou `Boolean`:
  retorna `true` se o próximo texto na entrada pode ser interpretado como um valor do tipo associado ao nome T.
- `t nextT()`, onde T deve ser substituído por `Int`, `Byte`, `Sort`, `Long`, `Double`, `Float` ou `Boolean`, e t por `int`, `byte`, `sort`, `long`, `double`, `float` ou `boolean`, respectivamente:
  retorna o valor do próximo *token* da entrada como um valor do tipo associado ao nome T.

Um esquema de código para leitura de valores primitivos a partir da entrada padrão `System.in` pode ter o seguinte modelo:

```
Scanner f = new Scanner(System.in);
long x = f.nextLong();
...
```

E se a entrada de dados for um arquivo de nome `a.txt`:

```
File entrada = new File("a.txt");
Scanner f = new Scanner(entrada);
long x = f.nextLong();
...
```

Para uma visão completa dos recursos oferecidos pela classse `Scanner`, recomenda-se a leitura dos tutoriais da Oracle.

9.4 Entrada e saída básicas

A implementação apresentada a seguir da biblioteca `myio`, que está descrita na Seção 1.10.2, ilustra o uso das classes básicas de entrada

e saída de arquivos de fluxos de *bytes* e de caracteres para prover um conjunto de operações de entrada/saída de alto nível.

9.4.1 Classe `Kbd.java`

Os métodos dessa classe permitem abrir o teclado para leitura e converter os caracteres digitados para o tipo indicado no seu nome. São ignorados os espaços em branco e caracteres ascii de controle que por ventura forem digitados antes da sequência de caracteres a ser lida. A sequência termina quando for digitado um caractere branco ou de controle.

```
1  package myio;
2  import java.io.*;
3  import java.util.*;
4
5  public class Kbd {
6   private static Scanner infile = new Scanner(System.in);
7   private static StringTokenizer st;
8
9   final static public boolean readBoolean()
10                                          throws IOException {
11    if (st == null || !st.hasMoreTokens())
12         st = new StringTokenizer(infile.nextLine());
13    return Boolean.valueOf(st.nextToken());
14  }
15  final static public int readInt() throws IOException {
16    if (st == null || !st.hasMoreTokens())
17        st = new StringTokenizer(infile.nextLine());
18    return Integer.parseInt(st.nextToken());
19  }
20  final static public long readLong() throws IOException {
21    if (st == null || !st.hasMoreTokens())
22        st = new StringTokenizer(infile.nextLine());
23    return Long.parseLong(st.nextToken());
24  }
```

```
25   final static public float readFloat() throws IOException {
26     if (st == null || !st.hasMoreTokens())
27         st = new StringTokenizer(infile.nextLine());
28     return Float.valueOf(st.nextToken());
29   }
30   final static public double readDouble() throws IOException {
31     if (st == null || !st.hasMoreTokens())
32         st = new StringTokenizer(infile.nextLine());
33     return Double.valueOf(st.nextToken());
34   }
35   final static public String readString() throws IOException {
36     if (st == null || !st.hasMoreTokens())
37         st = new StringTokenizer(infile.nextLine());
38     return st.nextToken();
39   }
40 }
```

9.4.2 Classe `Screen.java`

Os métodos da classe **`Screen`**, listados abaixo, convertem o parâmetro dado para uma sequência de caracteres ascii e a envia para o monitor de vídeo, para exibição a partir da posição corrente do cursor. Os métodos cujos nomes terminam por **`ln`** forçam uma mudança de linha após a escrita da sequência.

```
1  package myio;
2  public class Screen {
3    final static public void println() {
4       System.out.println();
5    }
6    final static public void print(String s) {
7       System.out.print(s);
8    }
9    final static public void println(String s) {
10      System.out.println(s);
11   }
```

```
12    final static public void print(boolean b) {
13       System.out.print(b);
14    }
15    final static public void println(boolean b) {
16       System.out.println(b);
17    }
18    final static public void print(byte b) {
19       System.out.print(b);
20    }
21    final static public void println(byte b) {
22       System.out.println(b);
23    }
24    final static public void print(char c) {
25       System.out.print(c);
26    }
27    final static public void println(char c) {
28       System.out.println(c);
29    }
30    final static public void print(int i) {
31       System.out.print(i);
32    }
33    final static public void println(int i) {
34       System.out.println(i);
35    }
36    final static public void print(long n) {
37       System.out.print(n);
38    }
39    final static public void println(long n) {
40       System.out.println(n);
41    }
42
43    final static public void print(float f) {
44       System.out.print(f);
45    }
46    final static public void println(float f) {
47       System.out.println(f);
48    }
```

```
49    final static public void print(double d) {
50       System.out.print(d);
51    }
52    final static public void println(double d) {
53       System.out.println(d);
54    }
55 }
```

9.4.3 Classe `InText.java`

Os métodos da classe **`InText`** funcionam como os de **`Kbd`**, exceto que os caracteres são lidos do arquivo cujo nome foi informado ao método construtor da classe, que é o responsável por tomar as providências para a abertura do arquivo.

```
1  package myio;
2  import java.io.*; import java.util.*;
3  import java.io.BufferedReader;
4  import java.io.FileReader;
5  import java.nio.charset.Charset;
6
7  public class InText {
8   private Charset charset = Charset.forName("ISO-8859-1");
9   private BufferedReader infile;
10  private StringTokenizer st = null;
11  private String s;
12  public InText(String fileName) throws IOException {
13    infile = new BufferedReader(
14             new FileReader(fileName,charset));
15  }
16  final public boolean readBoolean() throws IOException {
17    if (st == null || !st.hasMoreTokens())
18        st = new StringTokenizer(infile.readLine());
19    return Boolean.valueOf(st.nextToken());
20  }
```

```
21  final public byte readByte() throws IOException {
22    if (st == null || !st.hasMoreTokens())
23        st = new StringTokenizer(infile.readLine());
24    return Byte.parseByte(st.nextToken());
25  }
26  final public int readInt() throws IOException {
27    if (st == null || !st.hasMoreTokens())
28        st = new StringTokenizer(infile.readLine());
29    return Integer.parseInt(st.nextToken());
30  }
31  final public long readLong() throws IOException {
32    if (st == null || !st.hasMoreTokens())
33        st = new StringTokenizer(infile.readLine());
34    return Long.parseLong(st.nextToken());
35  }
36  final public float readFloat() throws IOException {
37    if (st == null || !st.hasMoreTokens())
38        st = new StringTokenizer(infile.readLine());
39    return Float.valueOf(st.nextToken());
40  }
41  final public double readDouble() throws IOException {
42    if (st == null || !st.hasMoreTokens())
43        st = new StringTokenizer(infile.readLine());
44    return Double.valueOf(st.nextToken());
45  }
46  final public String readQuotedString()
47                                      throws IOException {
48    if (st == null || !st.hasMoreTokens())
49        st = new StringTokenizer(infile.readLine());
50    s = st.nextToken();
51    return st.nextToken();
52  }
53  final public String readString() throws IOException {
54    if (st == null || !st.hasMoreTokens())
55        st = new StringTokenizer(infile.readLine());
56    return st.nextToken();
57  }
```

```
58   final public void close() throws IOException {
59     infile.close();
60   }
61 }
```

9.4.4 Classe `OutText.java`

Os métodos de **`OutText`** são análogos aos da classe **`Screen`**, exceto que a saída é redirecionada para um arquivo de fluxo, criado pela função construtora da classe.

```
1  package myio; import java.io.*;
2  public class OutText {
3    private PrintStream out;
4    public OutText(String fileName) throws IOException {
5      OutputStream fout = new FileOutputStream(fileName);
6      out = new PrintStream(fout);
7    }
8    final public void println() {
9       out.println();
10   }
11   final public void print(String s) {
12      out.print(s);
13   }
14   final public void println(String s) {
15      out.println(s);
16   }
17   final public void print(boolean b) {
18      out.print(String.valueOf(b));
19   }
20   final public void println(boolean b) {
21      out.println(b + "");
22   }
23   final public void print(byte i) {
24      out.print(String.valueOf(i));
25   }
```

```
26    final public void println(byte i) {
27       out.println(String.valueOf(i));
28    }
29    final public void print(short i) {
30       out.print(String.valueOf(i));
31    }
32    final public void println(short i) {
33       out.println(String.valueOf(i));
34    }
35    final public void print(int i) {
36       out.print(String.valueOf(i));
37    }
38    final public void println(int i) {
39       out.println(String.valueOf(i));
40    }
41    final public void print(long n) {
42       out.print(String.valueOf(n));
43    }
44    final public void println(long n) {
45       out.println(String.valueOf(n));
46    }
47    final public void print(float f) {
48       out.print(String.valueOf(f));
49    }
50    final public void println(float f) {
51       out.println(String.valueOf(f));
52    }
53    final public void print(double d) {
54       out.print(String.valueOf(d));
55    }
56    final public void println(double d) {
57       out.println(String.valueOf(d));
58    }
59    final public void close() trows IOException {
60       out.close();
61    }
62 }
```

9.5 Conclusão

Este capítulo apresentou apenas um subconjunto essencial das classes que implementam entrada e saída de fluxo de bytes, e dessas classes, somente as operações mais simples, mas que já são suficientes para desenvolver aplicações interessantes.

À medida que o leitor for aprofundando seus conhecimentos da linguagem Java, um estudo mais detalhado da biblioteca de classes para realizar entrada e saída de dados é imprescindível.

Exercícios

1. Reimplemente o pacote `myio` usando o método `split` da classe `String` para particionar sequências de caracteres no lugar de `Stringtonkenizer`.
2. Acrescente novos métodos `readString` a `Kbd` e a `InText` para ler sequências de caracteres delimitadas por um caractere passado como parâmetro.

Notas bibliográficas

O texto mais importante para estudar entrada e saída de fluxo é a página de Java mantida pela Oracle (`http://www.oracle.com`).

Além da especificação precisa de cada uma das classes de arquivos de fluxo, os tutorais são altamente esclarecedores.

Capítulo 10

Classes Aninhadas

Classes e interfaces podem ser **externas** ou **internas**. As externas são as definidas em primeiro nível, e as internas, também chamadas **aninhadas**, são as definidas dentro de outras estruturas. Como classes implementam tipos, e interfaces servem para especificar os serviços de um tipo, a ideia de aninhamento pode ser levada a tipos. Assim, tipos aninhados são classes ou interfaces que são membros de outras classes ou declarados locais a blocos de comandos. Qualquer nível de aninhamento é permitido.

Esse mecanismo de aninhamento é uma forma de controlar a visibilidade de tipos no programa. Ele permite fazer um tipo valer somente dentro de uma classe ou de um bloco de comandos. Esse recurso serve para melhor estruturar o programa, reduzindo a quantidade de nomes disponíveis a cada ponto do programa.

Classes aninhadas são o resultado da combinação do mecanismo de estrutura de blocos com programação baseada em classes, em que visibilidade está associada a encapsulação. Tipos aninhados em classes podem ter os mesmos modificadores de acesso (`public`, `protected`, `private` ou *pacote*) que membros normais de classe.

A compilação de um arquivo-fonte contendo classes externas e internas gera um arquivo com extensão `.class` para cada classe contida no arquivo, seja ela interna ou externa. Os *bytecodes* de classes internas são armazenados em arquivos cujos nomes têm

a estrutura `NomeClasseExterna$NomeClasseInterna.class`, e os das classes externas, `NomeClasseExterna.class`.

Classes aninhadas podem ser estáticas ou não-estáticas, e isso tem influência na sua capacidade de ter acesso a elementos do escopo envolvente. Interfaces aninhadas são sempre estáticas.

10.1 Classes aninhadas estáticas

Uma classe aninhada estática é como uma classe externa, exceto que sua acessibilidade externa é definida pelo seu modificador de acesso. Uma classe aninhada estática somente pode fazer referências a membros estáticos de sua classe envolvente. O exemplo abaixo mostra o aninhamento de uma classe estática de nome `Permissões` na classe externa `Conta`, sendo sua visibilidade declarada `public`.

```
1  public class Conta {
2     private long número; private long saldo;
3     public static class Permissões {
4        public boolean podeDepositar;
5        public boolena podeSacar, podeEncerrar;
6        public void verifiquePermissão(Conta c) {
7           ... c.número ...     // OK
8           ... this.número ...  // Erro
9        }
10    }
11    public Permissões obtémPermissão() {
12       return new Permissões();
13    }
14 }
```

Os atributos da classe envolvente `Conta` declarados na linha 2 não são acessíveis do corpo de classe estática `Permissões`: a tentativa de acesso mostrada na linha 8 provoca erro de compilação.

Uma classe aninhada pode ser usada localmente sem qualificações, como ocorre na linhas 11 e 12 da classe `Conta` acima. Referências a tipo aninhado estático fora da classe que o declara devem ser qualificadas pelo nome da classe envolvente, como abaixo.

```
1 class Banco {
2    Conta c = new Conta();
3    Conta.Permissões p1 = new Conta.Permissões();
4    void f() {
5       Conta.Permissões p2;
6       p2 = c.obtémPermissão();
7       if  (p1.podeDepositar && p2.podeSacar) ....
8    }
9 }
```

Os atributos públicos de `Permissões` são acessíveis em todo local em que `Conta.Permissões` o for, como ocorre na linha 7. Os modificadores de acesso de uma classe aninhada têm efeito apenas para o uso da classe fora de sua classe envolvente. Dentro da classe envolvente, todos os membros da classe aninhada são visíveis, independentemente de eles serem declarados com visibilidade `public`, `private`, `protected` ou *pacote*. Observe o seguinte programa:

```
1 public class A {
2    static class B {
3        private int x = 1000, int z = 2000;
4    }
5    static public void main(String[] args) {
6       B b = new B();
7       int y = b.z;  // OK
8       y += b.x;     // OK, embora b.x seja private
9       System.out.print("y = " + y);
10   }
11 }
```

A classe `D` a seguir tem erro de compilação devido à tentativa de acesso a membro privado fora de seu escopo, ocorrida na linha 11 de `D`.

```
1  class C {
2     public static class B {
3         private int x = 1000;
4         public  int z = 2000;
5     }
6  }
7  public class D {
8     static public void main(String[] args) {
9        C.B b = new C.B();  // OK
10       int y = b.z;   // OK
11       y += b.x;  // Errado: b.x inacessível aqui
12       System.out.print("y = " + y);
13    }
14 }
```

10.2 Classes aninhadas não-estáticas

Classe aninhada não-estática é um conceito distinto do de classe aninhada estática. Enquanto esta constitui-se essencialmente em um mecanismo de controle de visibilidade e escopo de tipos, a primeira, além desse mecanismo, oferece a possibilidade de acesso a elementos definidos no escopo envolvente. Isto inclui membros das classes envolventes e também variáveis ou classes locais a blocos envolventes.

Subclasses de classe aninhada não herdam esses privilégios de acesso de sua superclasse.

Classe aninhada não-estática cria a noção de **objetos envolventes** e **objetos envolvidos**. Dessa forma, objetos de classe aninhada precisam da referência do objeto envolvente, provido no

momento da criação do objeto, a partir da qual, pode-se acessar seus campos.

O exemplo a seguir é uma cópia ligeiramente modificada da classe `Conta` anteriormente apresentada. As modificações são a declaração da classe `Permissões` como aninhada **não-estática**, em vez de estática, e as mudanças decorrentes, especialmente na forma de criação de objetos de classe interna.

```
1  public class Conta {
2     private long número; private long saldo;
3     public class Permissões {
4        public boolean podeDepositar,
5                       podeSacar, podeEncerrar;
6        public void verifiquePermissão(Conta c) {
7           ... c.número ...      // OK
8           ... this.número ...   // OK
9        }
10    }
11    public Permissões obtémPermissão() {
12       return new Permissões();
13    }
14 }
15
16 class Banco {
17    Conta c = new Conta();
18    Conta.Permissões p1 = c.new Permissões();
19    void f() {
20       Conta.Permissões p2;
21       p2 = c.obtémPermissão();
22       if  (p1.podeDepositar && p2.podeSacar) ...
23    }
24 }
```

Destaca-se que nessa nova versão de `Conta`, a referência ao atributo `número` da classe envolvente, ocorrida na linha 8 de `Conta` agora está correta, e que, na linha 18, o operador `new` apresenta-

se qualificado pela referência ao objeto envolvente e o nome da construtora não é qualificado.

Isso ocorre porque, diferentemente do processo usual de criação de objetos de classes externas, a criação de objeto de classes internas não-estáticas exige a especificação do **objeto envolvente**, i.e., do objeto que contém os campos envolventes acessíveis a objetos da classe interna, no caso o objeto `c`.

Sintaticamente o novo operador `new` tem o formato:

```
x.new ClasseInterna(...)
```

onde `x` é uma referência ao objeto envolvente, e `ClasseInterna` é o nome, sem qualificação, da construtora da classe aninhada declarada diretamente dentro da classe do objeto ao qual `x` se refere. O uso de qualificação no nome da construtora de um `new` de classe aninhada não-estática não é permitido.

O endereço do objeto envolvente é armazenado no leiaute objeto criado e se essa referência for omitida no operador `new`, toma-se por *default* a referência ao objeto que está criando a instância da classe aninhada, i.e., o `this`.

O aninhamento de classe pode ser de qualquer nível. A partir de uma classe interna não-estática pode-se fazer referências sem qualificação a qualquer membro das classes envolventes.

Para resolver situações em que uma declaração torna oculta declarações em classes envolventes, porque usam um mesmo identificador, pode-se fazer referência explícita ao membro de classe desejado com a notação:

```
NomeDaClasseEvolvente.this.nomeDoMembro
```

O esquema de código abaixo, na definição da classe `X`, mostra como se fazem referências não-ambíguas a membros das classes envolventes em um dado aninhamento, por meio do uso de `this` devidamente qualificado pelo nome da classe que declara o atributo referenciado.

```
1  class X {
2     int a, d;
3     class Y {
4        int b, d;
5        class Z {
6           int c, d;
7           ... a ...          // X.this.a é o  a de X
8           ... b ...          // Y.this.b é o  b de Y
9           ... c ...          // Z.this.c é o  c de Z
10          ... X.this.d ...   // d de X
11          ... Y.this.d ...   // d de Y
12          ... d ...          // Z.this.d é o d de Z
13       }
14    }
15 }
```

A qualificação explícita de referências dentro da classe interna `Z` acima somente se faz necessária nas linhas 10 e 11, porque, por *default*, `d` da linha 12 é interpretado como `Z.this.d`, referindo-se ao `d` que está declarado mais próximo, dentro de `Z`.

Ainda na presença de classes internas, o identificador-padrão `this` continua denotando a referência ao objeto corrente. Por exemplo, o `this` dentro de uma função `f`, que foi chamada por meio de `h.f()`, tem o valor de `h`. Entretanto, a referência `Y.this`, que ocorre dentro da classe `Z` interna à classe `Y`, denota a referência ao objeto envolvente. Por exemplo, se `Y.this` ocorresse dentro de um método de um objeto que foi criado pelo comando `y.new Z()`, então `Y.this` seria um campo do objeto `Z` criado e teria o valor de `y`.

Classes aninhadas são definidas relativamente à classe envolvente, portanto elas precisam saber a referência do objeto envolvente. Para resolver isto, o compilador Java acrescenta ao leiaute dos objetos de classe aninhada uma referência ao objeto envolvente.

A referência ao objeto envolvente é passada implicitamente como um parâmetro extra da construtora. Assim, quando um objeto de classe aninhada não-estática é criado, o objeto envolvente é associado ao objeto, isto é, o objeto criado possui uma referência para o objeto da classe envolvente.

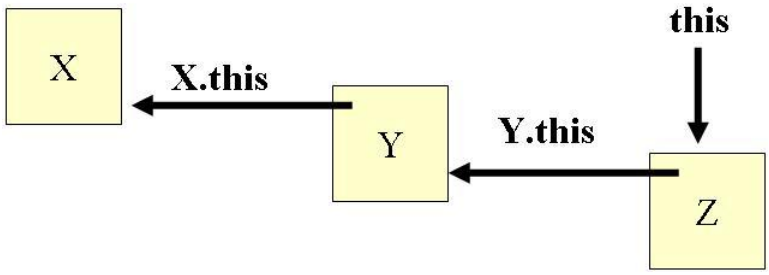


Figura 10.1 Objetos Envolventes

Figura 10.1 mostra o encadeamento de objetos envolvidos e envolventes do seguinte esquema de programa:

```
1  public class X {
2     int a, d;
3     public class Y {
4        int b, d;
5        public class Z {
6           int c, d;
7           void g() {
8              ... a ... b ... c ...
9              X.this.d ... Y.this.d ... this.d ...
10          }
11       }
12
13    }
14    public static void main(String[] args) {
15       A x = new X ();
16       X.Y y = x.new Y();
17       X.Y.Z  z = y.new Z();
18    }
19 }
```

10.3 Classes locais em blocos

Classes podem ser definidas dentro de blocos, inclusive dentro de métodos. Essas classes têm uso puramente local e têm acesso a variáveis declaradas finais e a membros da classe que contém o bloco. A restrição é que permite-se acesso somente a variáveis do bloco que sejam declaradas finais. Considere a título de exemplo, as seguintes declarações:

```
1  class FaltaElemento extends Exception { }
2  class Aluno {
3    private int n;
4    Aluno(int n) { this.n = n; }
5    int matricula() { return n; }
6  }
7  class CorpoDiscente {
8    private Object[] objetos;
9    public CorpoDiscente(Object[] objetos) {
10     this.objetos = objetos;
11   }
12   public Enumeração elementos() {
13     final int tamanho = objetos.length;
14     class EnumeraçãoLocal implements Enumeração {
15        private int posição = 0;
16        public boolean aindaHá(){return (posição<tamanho);}
17        public Object próximo()throws FaltaElemento {
18          if (posição >= tamanho)
19              throw new FaltaElemento();
20          return objetos[posição++];
21        }
22        public void reInicia() { posição = 0; }
23     }
24     return new EnumeraçãoLocal();
25   }
26 }
```

A interface **Enumeração** usada na classe **CorpoDiscente** tem a seguinte definição:

```
1 interface Enumeração {
2    boolean aindaHá();
3    Object próximo() throws FaltaElemento;
4    void reInicia() ;
5 }
```

As instâncias de classes locais são objetos normais, ou seja, podem ser retornados por funções, passados como parâmetros, etc. O uso desses objetos fora do bloco onde suas classes foram declaradas é possível se essas classes implementarem interfaces conhecidas externamente.

O programa **Acadêmico** apresentado a seguir usa as classes definidas acima e imprime `0 1 2 3 4 5 6 7 8 9`.

```
1  public class Acadêmico {
2    public static void main(String[] args) {
3      Aluno a, alunos[] = new Aluno[10];
4      for (int i = 0; i < alunos.length; i++)
5                                    alunos[i] = new Aluno(i);
6      CorpoDiscente c = new CorpoDiscente(alunos);
7      Enumeração e = c.elementos();
8      while (e.aindaHá()) {
9        try {
10          a = (Aluno) e.próximo();
11          System.out.print(a.matricula() + " ");
12       }
13       catch(FaltaElemento x) { }
14     }
15   }
16 }
```

Note que ao retornar da invocação de `c.elementos` da linha 7, o valor da variável local **tamanho** (linha 13 de **CorpoDiscente**), que

foi alocada no momento de entrada do método, não pode ser perdido, porque, ao se criar um objeto da classe `EnumeraçãoLocal`, cria-se uma entidade que faz, nas linhas 16 e 18, referências à variável local `tamanho` e que sobrevive à execução de `elementos`. Assim, variáveis locais, como `tamanho`, que são referidas por objetos de classe interna a método devem ser declaradas como `final`, para garantir que os seus valores sobrevivam ao retorno da função que as alocou.

Membros de classe envolvente, como `objetos` no exemplo acima, não precisam ser declarados finais, porque seu tempo de vida extrapola a dos dados alocados por seus métodos. No caso, o vetor `objetos` declarado na classe que contém o método `elementos` permanece acessível a partir do objeto `Enumeração` retornado.

10.4 Classes anônimas

Se uma classe for declarada para ser usada uma única vez para criação de um único objeto no local onde foi declarada, seu nome é desnecessário e pode ser omitido, declarando-se a classe diretamente como operando do operador construtor de objetos `new`. Esse tipo de classe é chamado de classe anônima.

Situação como essa ocorre com frequência na criação de ouvintes de eventos, onde a classe do ouvinte e o respectivo objeto aparecem somente uma vez no momento de associação do ouvinte ao componente gráfico gerador do evento. No Capítulo 19 esse recurso é usado com frequência. Outro exemplo é o programa principal padrão usado neste livro para criação de janelas gráficas.

A classe `CorpoDiscente` do exemplo anterior, que contém a classe `EnumeraçãoLocal`, declarada no método `elementos` e destacada no código abaixo, ilustra um bom uso de classes anônimas.

```
1  class CorpoDiscente {
2     private final Object[] objetos;
3     public CorpoDiscente(Object[] objetos) {
4        this.objetos = objetos;
5     }
6
7     public Enumeração elementos() {
8       final int tamanho = objetos.length;
9       class EnumeraçãoLocal implements Enumeração {
10         private int posição = 0;
11         public boolean aindaHá(){
12            return (posição<tamanho);
13         }
14         public Object próximo() throws FaltaElemento {
15           if (posição >= tamanho)
16               throw new FaltaElemento();
17           return objetos[posição++];
18          }
19          public void reInicia() { posição = 0; }
20       }
21       return new EnumeraçãoLocal();
22    }
23
24 }
```

O programa acima define o método **elementos** que essencialmente retorna um objeto do tipo **Enumeração** que percorre passo-a-passo uma lista de objetos definida pela classe **CorpoDiscente**.

A classe local **EnumeraçãoLocal** tem garantidamente um único uso, que é o da linha 21, como operando do operador **new**. Seu nome adiciona muito pouco à clareza do programa e pode ser eliminado do código, substituindo a chamada construtora, operando do **new**, por uma declaração de classe anônima, como mostra a nova versão do método **elementos** de **CorpoDiscente** apresentada a seguir.

```
1  public Enumeração elementos() {
2     final int tamanho = objetos.length;
3     return new Enumeração {
4        private int posição = 0;
5        public boolean aindaHá() {
6           return (posição < tamanho);
7        }
8        public Object próximo() throws FaltaElemento {
9           if (posição >= tamanho)
10              throw new FaltaElemento();
11          return objetos[posição++];
12       }
13       public void reInicia() { posição = 0; }
14       }
15    }
16 }
```

A expressão `new A(...){corpo da classe anônima}` tem o efeito de criar uma subclasse (ou uma implementação) de A com o corpo de classe anônima especificado.

Classes anônimas não podem ter construtoras, pois não há como especificar sua lista de parâmetros formais, uma vez que não têm nome. A lista de argumentos especificada no **`new`** é sempre passada à construtora da superclassse.

Se a classe anônima for derivada de uma interface `I`, a superclasse é `Object`, e a classe anônima é a implementação de `I`. Esse é o único contexto em que uma interface pode aparecer como operando de **`new`**. No caso de classe anônima que implementa interface, a lista de argumentos da construtora deve ser sempre vazia para ser compatível com a construtora da superclasse `Object`.

Classes anônimas compiladas recebem o nome interno

`NomeClasseExterna$`n`.class`,

onde n é um inteiro que representa o número de ordem da classe.

10.5 Extensão de classes internas

As classes aninhadas estáticas podem ser estendidas como qualquer classe de primeiro nível ou externa. As classes internas não-estáticas também podem ser estendidas, mas deve-se cuidar para que os objetos da classe estendida sejam devidamente associados a objetos da classe envolvente à classe sendo estendida.

A classe estendida pode ou não ser uma classe interna em um ou mais níveis da classe envolvente. O objeto envolvente dessa classe estendida é o mesmo de sua superclasse, conforme ilustra o seguinte exemplo:

```
1  class A {
2     class B { }
3  }
4
5  class C extends A {
6     class D extends B {  }
7  }
8
9  public class Main {
10    public static void main(String[] args) {
11       A a = new A();
12       A.B b1 = a.new B();
13       C c = new C();
14       C.B b2 = c.new B();
15       C.D  d = c.new D();
16    }
17 }
```

No caso de a classe estendida não ser relacionada com a classe envolvente, deve-se prover o objeto envolvente quando o construtor da superclasse for invocado via `super`, tal qual é feito na linha 6 do código abaixo, para executar a construtora de `A.B` para o objeto envolvente `a`, passado como parâmetro.

```
1  class A {
2     class B { }
3  }
4
5  class D extends A.B {
6     D(A a) { a.super() }
7  }
8
9  public class Main {
10    public static void main(String[] args) {
11       A a   = new A();
12       A.B b = a.new B();
13       D d   = new D(a);
14    }
15 }
```

10.6 Conclusão

Classes aninhadas são o resultado da combinação do mecanismo de estrutura de blocos da programação tradicional com a programação baseada em classes, em que visibilidade está associada a encapsulação.

Esse recurso permite controlar melhor a encapsulação de tipos ao longo do programa, eliminando da preocupação do programador nomes que não interessam em um dado momento.

Aninhamento de classes também é muito útil na implementação de ouvintes de eventos, os quais normalmente precisam de ter acesso a dados do componente que gera os eventos que devem ser tratados. A declaração da classe do ouvinte internamente à classe do componente originador de eventos facilita esse processo.

Exercícios

1. Qual é a principal diferença entre classes aninhadas estáticas e classes aninhadas não-estáticas?
2. Dê um exemplo em que o uso de classe anônima é recomendado.
3. Qual é o papel o objeto envolvente?

Notas bibliográficas

A melhor referência bibliográfica para conhecer em profundidade os mecanismos de classes aninhadas é a página oficial da Oracle para a linguagem Java (http://www.oracle.com).

O livro de Ken Arnold et alii [3] é também uma referência importante e de fácil leitura.

Capítulo 11

Pacotes

Pacote (`package`) é um recurso para agrupar física e logicamente tipos[1] relacionados, sendo dotado de mecanismo próprio para controle de visibilidade. Pacote corresponde a uma pasta do sistema de arquivos do sistema operacional. Pacote, portanto, é uma coletânea de arquivos, cada um contendo declarações de tipos, e possivelmente contendo outros pacotes.

Cada arquivo em um pacote pode ter no máximo a declaração de um tipo público, o qual pode ser usado localmente ou em outros pacotes. As demais classes ou interfaces não-públicas, declaradas no arquivo são estritamente locais e visíveis apenas dentro do pacote. Pacotes servem para criar bibliotecas de classes e interfaces.

Uma unidade de compilação em Java é um arquivo que contém o código fonte de uma ou mais classes e interfaces. Se o arquivo contiver a declaração de um tipo público, o nome desse arquivo deve ser o mesmo do tipo público declarado, acrescido da extensão `.java`. Cada arquivo-fonte de um pacote pode ser iniciado por uma instrução `package`, que identifica o pacote, isto é, a biblioteca à qual os *bytecodes* de suas classes ou interfaces pertencem. Logo após, no código-fonte, têm-se zero ou mais instruções `import`, que têm a finalidade de informar ao compilador a localização dos tipos de outros pacotes usados nessa unidade de compilação.

[1] Tipos em Java são classes ou interfaces

O pacote `java.lang`, por ser de uso frequente, dispensa importação explícita: seus tipos são automaticamente importados para toda unidade de compilação.

Para ilustrar a organização de pacotes, suponha que a pasta `figuras` contenha os seguintes arquivos:

- Arquivo `Forma.java`:

```
package figuras;
public class Forma { ... }
definição de outros tipos não-públicos
```

- Arquivo `Retângulo.java`:

```
package figuras;
public class Retângulo { ... }
definição de outros tipos não-públicos
```

- Arquivo `Círculo.java`:

```
package figuras;
public class Círculo { ... }
definição de outros tipos não-públicos
```

Em alguma outra pasta de seu sistema de arquivo, pode-se utilizar as classes públicas do pacote `figuras`, por exemplo, nas declarações de tipos do arquivo `ProgramaGráfico.java`, o qual deve ter a seguinte estrutura:

```
1 import figuras.*;
2 public class ProgramaGráfico {
3     ...
4     usa os tipos importados Forma, Rectângulo e Círculo
5     ...
6 }
```

onde a diretiva de compilação `import figuras.*` provê informações para o compilador localizar a pasta `figuras`, que contém os arquivos `Forma.class`, `Rectângulo.class` e `Círculo.class`, usados no programa.

11.1 Importação de pacotes

Os tipos declarados como não-públicos em um pacote são visíveis apenas nas classes ou interfaces do mesmo pacote. Os tipos declarados públicos podem ser usados em outros pacotes.

Para se ter acesso a tipos públicos de outros pacotes, o compilador deve conhecer o nome completo dos pacotes usados. O nome completo de um pacote é o seu *caminho de acesso* desde a raiz da árvore de diretórios de seu sistema de arquivos até a pasta que lhe corresponde.

Uma forma de se ter acesso às classes de outros pacotes é informar explicitamente ao compilador, a cada uso de um tipo, o seu nome completo. Por exemplo, a declaração

```
C:\MeusProgramas\Java\Ed.Pilha p
```

define `p` como uma referência para objetos do tipo `Pilha` que está definido na pasta[2] `C:\MeusProgramas\Java\Ed`.

Entretanto, a especificação de caminhos absolutos, como o mostrado acima, torna a aplicação muito inflexível e de difícil transporte para outros computadores, mesmo quando operam sob o mesmo sistema operacional. Java resolve esse problema por meio da instrução `import` e da variável de ambiente `CLASSPATH`, que permitem tornar a especificação de caminho independente do computador em que o pacote estiver instalado. No programa apenas o nome do tipo é usado. O seu caminho de acesso passa a ser obtido da variável de ambiente `CLASSPATH` e da instrução `import`, na forma descrita a seguir.

A variável de ambiente `CLASSPATH` deve ter seu valor definido pelo programador para cada computador ou plataforma em uso com os prefixos dos caminhos possíveis para se chegar aos pacotes

[2] O caractere "\" representa o símbolo delimitador de pasta usado na especificação de caminhos no sistema de arquivos. Há sistemas operacionais que usam "/"no lugar de "\".

desejados. Por exemplo, se em uma dada plataforma conhecida, a variável de ambiente `CLASSPATH` for definida por

```
set CLASSPATH = .;C:\Java\Lib;C:\projeto\fonte
```

então os pacotes Java usados nos programas nessa plataforma devem estar localizadas dentro da pasta corrente, ou da pasta `C:\Java\Lib` ou da pasta `C:\projeto\fonte`. A busca pelos tipos usados no programa é feita pelo compilador nessa ordem.

`CLASSPATH` dá o prefixo dos caminhos possíveis, e o comando `import` permite adicionar trechos de caminhos no fim desses prefixos para formar o caminho completo, o qual permite ao compilador encontrar os arquivos `.class` de tipos usados no pacote e que não foram definidos localmente. Por exemplo, `import A.t`, no contexto dos valores de `CLASSPATH` definidos acima, faz os possíveis caminhos completos do tipo `t`, o arquivo `t.class`, externo ao pacote, serem: `.\A`, `C:\Java\Lib\A` ou `C:\projeto\fonte\A`.

Suponha a compilação de um arquivo `A.java`, contendo a instrução `import x.y.z.t`. Para cada ocorrência do tipo `t` que não seja declarado localmente, o compilador segue os seguintes passos:

1. pega nome completo da classe, incluindo nome do seu pacote, conforme definido pela instrução `import`, e.g., `x.y.z.t`;
2. substitui os pontos (`"."`) por separadores de diretórios (`"/"` ou `"\"`), conforme a plataforma, e.g., `x\y\z\t` ;
3. acrescenta sufixo `.class` a esse caminho, e.g., `x\y\z\t.class`;
4. prefixa o resultado com os valores definidos no `CLASSPATH`, produzindo no presente exemplo os caminhos:

   ```
   .\x\y\z\t.class
   C:\Java\Lib\x\y\z\t.class
   C:\projeto\fonte\x\y\z\t.class
   ```

Os caminhos obtidos acima são os usados pelo compilador Java para se ter acesso ao arquivo do `bytecode` da classe desejada.

O comando `import` apenas indica parte do caminho de acesso aos tipos externos a um pacote. Essa indicação pode ser específica, tipo por tipo, ou genérica, designando vários tipos.

Por exemplo, para usar os tipos `t` e `u` definidos em um pacote X sem qualquer especificação de caminho, pode-se fazer suas importações individuais, por exemplo `import X.t; import X.u`, ou então coletivamente por meio do comando `import X.*`. Somente os tipos usados são efetivamente incorporados ao código. Por ser de uso frequente, todas as compilações importam os tipos do pacote `java.lang.*` implicitamente.

11.2 Resolução de conflitos

Durante a compilação de um arquivo-fonte, a descrição de cada tipo não-primitivo usado deve ser fornecida ao compilador, para tornar possível as verificações quanto à correção de seu uso no programa. Assim para cada tipo presente no fonte, o compilador deve realizar uma busca no sistema de arquivo para localizar os arquivos `.class` desejados.

Essa busca por um tipo `t` encontrado no programa-fonte obedece à seguinte ordem:

1. Tipo `t` é o tipo aninhado no tipo sendo compilado. Por exemplo, o tipo `A` usado na linha 4 é o da linha 7 no programa:

```
1 import X.*;
2 class A { void f( ) { } }
3 class B extends A {
4    A a = new A( );   // A é o da linha 7
5    a.g( );                    // OK
6    a.f( );                    // Erro
7    class A { void g( ) { } }
8 }
```

2. Tipo `t` é o tipo que está sendo compilado, incluindo tipos herdados. A busca é feita dentro da hierarquia de tipos. Por exemplo, as ocorrências de `A`, `B` e `C` durante a compilação da classe `C` no código

```
1 import X.*;
2 class A {  ... }
3 class B extends A {  ... }
4 class C extends B { ... A ... B ... C ... }
```

   são dos tipos declarados nas linhas 2, 3 e 4, respectivamente, mesmo se o pacote `X` contiver tipos com os mesmos nomes.
3. Tipo `t` é o tipo importado com qualificação completa, e.g., `import X.t`.
4. Tipo `t` é tipo declarado no mesmo pacote.
5. Tipo `t` é um tipo importado por `import X.*`.

Conflitos de nomes ocorrem quando houver importação individual de tipos com mesmo nome de mais de um pacote ou importação individual de um tipo e declaração local de outro tipo com mesmo nome. Nesses casos, deve-se resolver os conflitos usando-se qualificação completa em toda referência aos tipos conflituosos.

11.3 Visibilidade

Há em Java três níveis de visibilidade a elementos de um programa: nível de classe, nível de pacote e nível de módulos. Pacote é essencialmente um dos mecanismos que permitem agrupar e delimitar a visibilidade de tipos e membros de tipo.

Tipos não-aninhados que são declarados dentro de pacotes podem ser de acesso público, i.e., podem ser usados fora do pacote, ou acesso pacote, ou seja, uso apenas dentro do pacote. Esses tipos não podem ser declarados `protected` ou `private`: ou especifica-

se acesso público pelo modificador de acesso `public`, o que torna o tipo acessível fora do pacote, ou então usa-se acesso pacote, para restringir o acesso a tipos locais do pacote. Acesso pacote é obtido pela ausência de modificador de acesso ao tipo.

No nível de classes, membros podem ser declarados `private`, `public`, `protected` ou *pacote* (sem especificação de modificador de acesso). Membros privados têm visibilidade restrita à sua classe. Membros públicos são visíveis onde sua classe for visível. Membros protegidos tem visibilidade em qualquer tipo do mesmo pacote e em subclasses de sua classe. Membros declarados com visibilidade *pacote* são visíveis apenas dentro do pacote que o contém. A visibilidade a tipos definida por módulos está detalhada no Cap.12.

O exemplo a seguir mostra a visibilidade de classes e de diversos atributos e métodos das classes de dois pacotes, denominados `BibliotecaA` e `BibliotecaB`, onde alguns membros têm visibilidade pública e outros a têm confinada nas fronteiras do pacote onde foram declarados.

- Arquivo `A.java` dentro da pasta `BibliotecaA`:

```
1  package BibliotecaA;
2  public class A {           // visível pacote e fora
3    private int pri = 1;    // visível somente classe A
4    public int pub = 2;     // visível onde A for
5    protected int pro = 3; // visível pacote e subs
6    int pac = 4;            // visível no pacote
7  }
8  class B {                  // visível no pacote
9    private int pri = 1;    // visível classe B
10   public int pub = 2;     // visível pacote
11   protected int pro = 3; // visível pacote
12   int pac = 4;            // visível pacote
13 }
```

- Arquivo `F.java` dentro da pasta `BibliotecaA`:

```
1  package BibliotecaA;
2  public class F {
3     public static void main(String[] args) {
4        int x;
5        A a = new A( );                  // OK
6        B b = new B( );                  // OK
7        x = 1 + a.pub + b.pub;           // OK
8        x = x + a.pro + b.pro;           // OK
9        x = x + a.pac + b.pac;           // OK
10       System.out.println("X = " + x);
11    }
12 }
```

- Arquivo `G.java` dentro da pasta `BibliotecaB`:

```
1  package BibliotecaB;
2  import BibliotecaA.*;
3  public class G {
4     public static void main(String[] args) {
5        int x;
6        A a = new A( );     // OK
7        B b = new B( );     // Erro de compilação
8        x = 1 + a.pub ;     // OK
9        x = x + a.pro ;     // Erro de compilação
10       x = x + a.pac ;     // Erro de compilação
11       System.out.println("X = " + x);
12    }
13 }
```

11.4 Compilação com pacotes

O processo de compilação cria, para cada classe declarada no arquivo-fonte, arquivos *bytecode*, cujos nomes são os das classes nele

contidas e todos com extensão `.class`.

Uma forma de ativar o compilador Java é o comando de linha:

`javac [ -classpath` *caminhos* `]` *arquivos-fonte*

onde *arquivos-fonte* é uma lista de um ou mais arquivos a serem compilados, por exemplo, `A.java B.java`, e *caminhos*, os argumentos da diretiva `-classpath`, especificam onde encontrar os arquivos `.class` usados. A diretiva `-classpath` substitui os caminhos de acesso às classes especificadas pela variável de ambiente `CLASSPATH`. Se `-classpath` e `CLASSPATH` não forem especificados, somente as classes que estiverem no diretório corrente podem ser usadas. Se apenas `-classpath` não for especificado, o diretório corrente e os definidos pela variável de ambiente `CLASSPATH` serão pesquisados.

O compilador exige que todo tipo referenciado no pacote deve ter sido nele definido ou então importado de outros pacotes. A localização desses pacotes deve ser informada ao compilador pela combinação das informações de `-classpath` (ou `CLASSPATH`) e do comando `import`.

Nos exemplos apresentados a seguir, supõe-se que o caractere `"\"` denote o símbolo usado na especificação de caminhos no sistema de arquivo, e que `"X>"` indique que a linha de comando é disparada a partir da pasta `X`. Supõe-se também que a variável de ambiente `CLASSPATH` não tenha sido especificada.

Ao se compilar uma classe `A.java`, que esteja dentro de uma pasta `X` e que usa tipos que podem estar na própria pasta `X`, na pasta ascendente de `X` ou na pasta `X\D1\D2`, deve-se usar a diretiva `-classpath` para instruir o compilador sobre esses locais da forma apresentada a seguir:

```
X>javac -classpath .;..;D1\D2 A.java
```

Os operandos de `-classpath` são definidos tendo como referência a pasta `X`, de onde a compilação é disparada. O caractere

";" separa as opções de caminhos.

Na definição do caminho a informar ao compilador, deve-se considerar que se o arquivo que contém a definição de uma classe `A` iniciar-se com `package P`, o nome desta classe é `P.A`, e sua localização é o diretório que contém a pasta `P`.

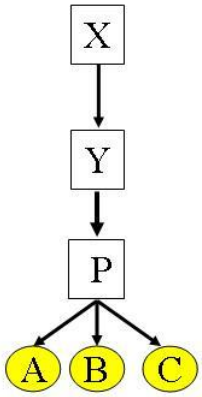


Figura 11.1 Hierarquia de Pacotes I

Considere a hierarquia de diretórios, mostrada na Fig. 11.1, cujas folhas são os três arquivos esquematizados no trecho de código abaixo:

```
1 // Arquivo A.java:
2    package P; public class A { ... }
3 // Arquivo B.java:
4    package P; public class B { ... }
5 // Arquivo C.java:
6    package P; public class C {  ... A ... B ... }
```

Suponha que as classes `A` e `B` somente usem tipos básicos, e que a classe `C` use somente os tipos `A` e `B`, conforme indicado acima. Assim, para compilar `A` e `B`, a partir de `P`, não é necessário a diretiva `-classpath`, porque estas classes não usam tipos externos ao pacote `P`, basta comandar:

```
P>javac  A.java
P>javac  B.java
```

Entretanto, para a compilação do programa em `C.java`, deve-se usar a diretiva `-classpath`, conforme mostram as linhas de comando abaixo que tratam de disparos de compilação a partir das pastas `P`, `Y` e `X`, respectivamente:

- A partir de `P`:

  ```
  P>javac -classpath .. C.java
  ```

  A diretiva `-classpath` indica o local, relativo a `P`, onde estão as definições das classes `A` e `B`.

- A partir de `Y`:

  ```
  Y>javac -classpath . P\C.java
  ```

  Note que agora as classes `A.class` e `B.class` estão no diretório corrente da compilação e o arquivo `C.java` está no subdiretório `P` de `Y`.

- A partir de `X`:

  ```
  X>javac -classpath Y Y\P\C.java
  ```

  Nesse caso, a partir de `X`, encontram-se `A.class` e `B.class` no subdiretório `Y`, informado pelo `-classpath`, e o arquivo `C.java` encontra-se no subdiretório `P` que está dentro de `Y`, conforme especificado na linha de comando. A localização do arquivo-fonte a ser compilado não depende do `-classpath`.

O comando para executar o `main` da classe `P.C` a partir da pasta `X` é o seguinte:

```
X>java -classpath Y P.C
```

onde `-classpath Y` indica onde, a partir de `X`, os arquivos `A.class`, `B.class` e `C.class` estão armazenados.

Note que a localização da classe principal, `P.C`, é dada pela diretiva `-classpath`. Assim, no comando `java` somente seu nome completo, `P.C`, precisa ser fornecido, pois ele será completado para `Y\P\C.java`.

Para ilustrar uma compilação envolvendo mais de um pacote, considere a hierarquia de diretórios ou pastas mostrada na Fig. 11.2, onde as classes `A` e `B` estão no pacote `P`, e a classe `C` está no pacote `Q`.

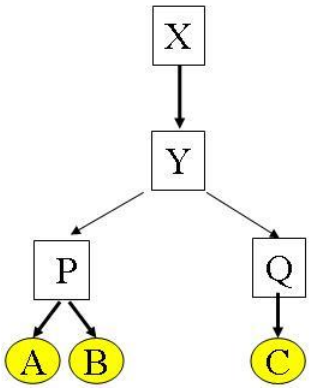


Figura 11.2 Hierarquia de Pacotes II

Uma implementação da Fig. 11.2 é a seguinte:

- Arquivos `A.java`:
```
package P;
public class A { ... }
```
- Arquivos `B.java`:
```
package P;
public class B { ... }
```
- Arquivos `C.java`:
```
package Q;
public class C {  ... A ... B ... }
```

Os comandos para disparar a compilação, dependendo de sua localização no sistema de arquivos, são:

```
P>javac  A.java
P>javac  B.java
Q>javac -classpath .. C.java
X>javac -classpath Y Y\Q\C.java
```

Observe que a partir de `X`, na quarta linha acima, `Y` dá a localização de `P.A` e `P.B` e que o caminho desde `X` a `P.C` deve ser explícito.

O nome de um pacote pode ser repetido, da mesma forma que um subdiretório pode ter o mesmo nome de um outro que não seja seu *irmão* na hierarquia, conforme exemplificado na Fig. 11.3, onde há duas pastas de nome `P`, que definem uma única biblioteca de nome `P`, contendo as classes `A`, `B` e `M`.

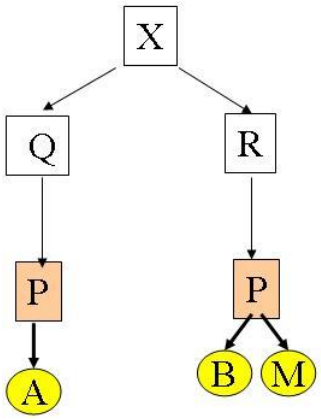


Figura 11.3 Hierarquia de Pacotes III

Suponha que os arquivos fontes que aparecem na Fig. 11.3 sejam:

- Arquivo `A.java`:

```
package P;
class A { int x; ... }
```

- Arquivo `B.java`: `X\R\P`:

```
package P;
class B { int y; ... }
```

- Arquivo `A.java`:

```
package P;
class M { ... A ... B... x ... y ... }
```

Os comandos de compilação desses arquivos a partir dos diretórios indicados abaixo são os seguintes:

```
X\Q\P>javac A.java
X\R\P>javac B.java
X\R\P>javac -classpath ..;..\..\Q M.java
```

A execução do `main` de `P.M` a partir da posição indicada no sistema de diretório é diparada por:

```
X\R\P>java -classpath ..;..\..\Q P.M
```

onde a diretiva `-classpath` indica as localizações relativas de `A.class` e `B.class`.

11.5 Conclusão

O ditado americano *boas cercas fazem bons amigos* aplica-se como uma luva à programação modular, que consiste em delimitar com clareza a visibilidade dos nomes definidos no programa.

Nesse sentido, Java provê as ferramentas necessárias para construção de boas cercas por meio de classes, pacotes e módulos, que são todos mecanismos que permitem agrupar e delimitar a visibilidade de tipos e membros de tipos.

Exercícios

1. Quais são os mecanismos de Java para controle de visibilidade de um tipo declarado no programa?

Notas bibliográficas

A melhor referência bibliográfica para conhecer em profundidade a estrutura de pacotes em Java é a página oficial da Oracle para a linguagem Java (http://www.oracle.com).

O livro de Ken Arnold et alii [3] é também uma referência importante e de fácil leitura.

Capítulo 12

Módulos

Programas em Java podem ser organizados como um conjunto de pacotes, os quais contêm classes, interfaces e possivelmente outros pacotes. Cada pacote de um programa, tipicamente, define, para uso local, um conjunto de tipos, enquanto exporta outros para uso em outros pacotes.

Em Programação Modular, módulo é definido com uma unidade de programa que pode ser compilada separadamente. Em Java, a menor unidade que pode formar essa categoria de módulo é a definição de um tipo de dados, que é uma classe ou uma interface. Módulos maiores podem ser construídos usando pacotes que agrupam várias definições de tipos e até outros pacotes.

Desse ponto de vista, Java é uma linguagem altamente modular. O mecanismo de exportação oferecido por pacotes, de fato, é poderoso e muito útil para estruturação de programas de maior porte. Entretanto, ele é excessivamente permissível, pois todos os tipos de dados exportados por um pacote são disponibilizados a todos os demais pacotes do programa, inclusive àqueles que não tenham qualquer relação com o pacote exportador. Essa permissividade pode, em princípio, ser considerada um potencial poluidor do ambiente de nomes de tipos disponíveis em certos contextos do programa.

12.1 Construção `module`

Java introduziu em suas versões mais recentes uma nova construção denominada `module` como uma ferramenta especial para agrupar pacotes que se relacionam e controlar a visibilidade dos nomes de tipos por eles exportados.

A construção `module` de Java nunca contém trechos de programa, como expressões, comandos, declarações, classes ou interfaces, mas apenas diretivas que descrevem e definem escopo de nomes de tipos de dados e as dependências existentes entre os grupos de pacotes de um programa.

Assim, para dar ao programador capacidade de disciplinar a visibilidade de nomes em um programa, Java oferece três recursos linguísticos:

- `class`: um contêiner de campos e métodos, sendo assim uma construção sintática que encapsula campos e métodos que podem ser públicos, protegidos ou privados.
- `package`: um contêiner de classes e interfaces, que podem ser públicas ou locais ao contêiner. Na prática, cada pacote corresponde a uma pasta ou diretório do sistema de arquivos em uso e geralmente exporta um conjunto de tipos nele definidos.
- `module`: um contêiner de pacotes, construído com a definição de quais são os pacotes que o compõem, qual a visibilidade externa de seus componentes e de quais outros módulos eles importam nomes.

O corpo de um `module` nunca contém fisicamente declarações de pacotes ou de seus elementos: ele apenas define relações de dependência entre pacotes. Essa construção, apesar do nome, não é o mesmo conceito clássico de módulo descrito anteriormente, pois, na prática, um `module` é apenas uma pasta ou diretório do sis-

tema de arquivos que armazena pacotes e a qual está associado em esquema de visibilidade aos seus constituintes.

Nessa organização de pacotes, o diretório de cada módulo contém subpastas que contêm pacotes, formando uma árvore de pacotes acrescida de um arquivo especial de nome `modulo-info.java`, necessariamente localizado na raiz dessa árvore, e que serve para definir a interface do módulo, explicitando os pacotes por ele requisitados, exportados, abertos ou ocultados.

12.2 Módulos e pacotes

Considere um programa denominado `EXPORTS`, que implementa os pacotes listados a seguir, cada um exportando uma classe para uso em outros pacotes:

- `p1a`: exporta classe `A1.java`
- `p2a`: exporta classe `A2.java`
- `p3a`: exporta classe `A3.java`
- `p1b`: exporta classe `B1.java`
- `p2b`: exporta classe `B2.java`
- `p1c`: exporta classe `C1.java`
- `p2c`: exporta classe `C2.java`
- `principal`: exporta classe `Main.java`

As classes exportadas pelos pacotes acima estão automaticamente disponíveis a todos eles, mas suponha que uma análise do programa `EXPORTS` permita disciplinar a visibilidade de tipos em cada pacote por meio da sua organização em quatro regiões de visibilidade ou módulos:

- `ma`: agrupa os pacotes `p1a`, `p2a` e `p3a`, que exportam um conjunto de tipos.

- `mb`: agrupa os pacotes `p1b` e `p2b`, que também exporta um conjunto de tipos.
- `mc`: agrupa os pacotes `p1c` e `p2c`, que usa tipos definidos nos pacotes de `ma`.
- `teste`: agrupa o pacote `principal`, que usa tipos exportados por pacotes de `mb` e `mc`.

E suponha que a dependência entre esses módulos seja a mostrada na Fig. 12.1, onde indica-se que `teste` usa nomes exportados por `mb` e `mc`, e módulo `mc` depende de tipos definidos em `ma`.

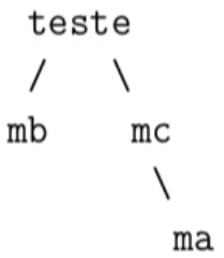


Figura 12.1 Dependência dos módulos

O mecanismo de modularização de pacotes de Java é baseado na associação de arquivos denominados `module-info.java` a cada um dos módulos definidos para o programa, de forma a definir as exportações e importações por eles realizadas, como exemplificado na Fig. 12.2.

A diretiva `exports` de um módulo define os pacotes que ele torna visíveis externamente, e a diretiva `requires` torna os nomes exportados pelo pacote especificado visíveis no módulo importador.

Observe que, conforme a Fig. 12.2, no módulo `teste` são visíveis os nomes exportados por `mb` e `mc`, e o módulo `ma` exporta apenas os tipos definidos em `p1a` e `p2a` e restringe a visibilidade dos nomes exportados pelo pacote `p3a` ao próprio `ma`, por não citá-lo em diretiva `exports`.

`module-info.java` de `ma`:

```
1 module ma {
2     exports ma.p1a;
3     exports ma.p2a;
4 }
```

`module-info.java` de `mb`:

```
1 module mb {
2     exports mb.p1b;
3     exports mb.p2b;
4 }
```

`module-info.java` de `mc`:

```
1 module mc {
2     requires ma;
3     exports mc.p1c;
4     exports mc.p2c;
5 }
```

`module-info.java` de `teste`:

```
1 module teste {
2     requires mb;
3     requires mc;
4 }
```

Figura 12.2 Módulos `module-info.java`

O programa **EXPORTS** deve ser armazenado na pasta de mesmo nome, conforme mostra a Fig. 12.3, a qual exibem duas subpastas: **src**, que contém as subpastas **ma**, **mb**, **mc** e **teste**, e a pasta **bin**.

As subpastas de **src** armazenam os pacotes dos módulos **ma**, **mb**, **mc** e **teste**, e o arquivo **bin** é o repositório, inicialmente vazio, dos *bytecodes* das classes e interfaces de **src**.

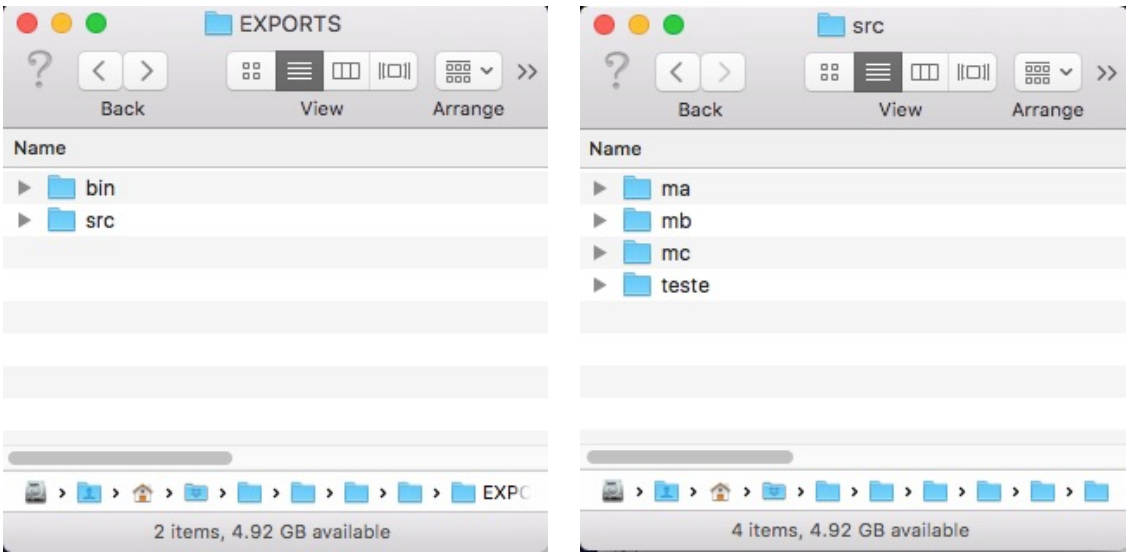


Figura 12.3 Pastas do programa EXPORTS

12.2.1 Compilação de `ma`

O módulo **`ma`** é uma subpasta da pasta **`EXPORTS`** e agrupa os pacotes **`p1a`**, **`p2a`** e **`p3a`** e arquivos auxiliares, mostrados na Fig.12.4, sendo o arquivo **`module-info.java`** da Fig. 12.2 o que define os pacotes exportados por **`ma`**, no caso, **`ma.p1`** e **`ma.p2a`**, enquanto restringe a terem apenas acesso local os tipos definidos em **`p3a`**.

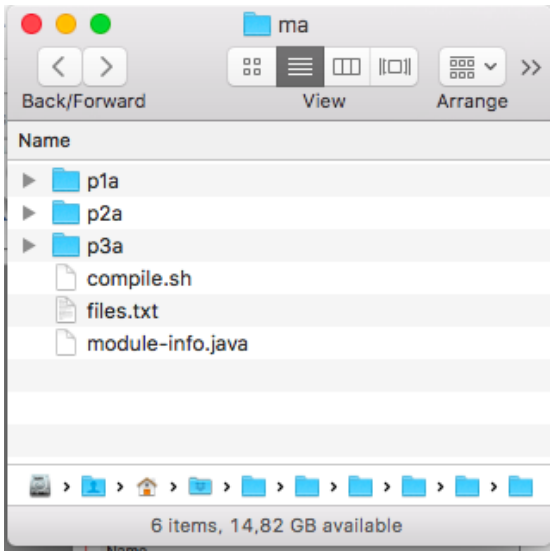


Figura 12.4 Módulo `ma`

Os diretórios **`p1a`**, **`p2a`** e **`p3a`** contêm, respectivamente, os seguintes arquivos **`A1.java`**, **`A2.java`** e **`A3.java`**:

- Arquivo `A1.java` do pacote `ma.p1a`:

```
1  package ma.p1a;
2  public class A1 {
3      public static String name() { return "A1"; }
4  }
```

- Arquivo A2.java do pacote `ma.p2a`:

```
1  package ma.p2a;
2  public class A2 {
3      public static String name() { return "A2"; }
4  }
```

- Arquivo A3.java do pacote `ma.p3a`:

```
1  package ma.p3a;
2  public class A3 {
3      public static String name() { return "A3";}
4  }
```

O arquivo *shell* `compile.sh` contém o comando Unix

```
javac -d ../../bin/ma @files.txt
```

que serve para disparar a compilação dos arquivos-fonte Java identificados no arquivo `files.txt` e que especifica a pasta na qual os *bytecodes* gerados pelo `javac` devem ser armazenados. Os caminhos especificados consideram que a compilação será disparada a partir da pasta `ma`.

O arquivo `files.txt` usado pelo comando especificado no *shell* `compile.sh` contém os nomes dos pacotes do módulo `ma` a ser compilados e do arquivo `module-info.java` que especifica de sua interface:

```
module-info.java
p1a/A1.java
p2a/A2.java
p3a/A3.java
```

Observe que o arquivo `module-info.java` da pasta `ma` deve obrigatoriamente fazer parte de `files.txt`, pois é nele que se especificam as dependências de outros módulos.

A forma mais geral do comando `javac` permite informar o local dos módulos importados pelos arquivos fontes a ser compilados. Por exemplo, supondo que o `javac` esteja sendo executado no seguinte contexto:

```
EXPORTS
 |__ bin    <-- ../../bin local para armazenar os binários
 |__ src
    |__ ma  <-- execução do javac de dentro de ma
    |__ ...
```

compila-se o módulo `ma` com o comando:

`javac` `--module-path ../../bin` (módulos usados) -d ../../bin/ma (binários) `@files.txt` (arquivos .java)

12.2.2 Binários de `ma`

O código compilado do programa `EXPORTS` fica armazenado na pasta `EXPORTS/bin`, na qual há subpastas de binários para cada módulo que compõe a aplicação. A estrutura desses diretórios faz parte da organização dos módulos. Em particular, os *byte-codes* dos pacotes que compõem o módulo `ma` são armazenados em `EXPORTS/bin/ma`, conforme mostrado em detalhes na Fig. 12.5.

A pasta `EXPORTS/bin/ma` armazena o corresponente arquivo `module-info.class` e uma subpasta também chamada `ma`, onde efetivamente residem os binários dos pacotes desse módulo. Assim, dentro de `EXPORTS/bin/ma/ma`, estão as pastas dos binários dos pacotes `ma.p1a`, `ma.p2a` e `ma.p3a`.

12.2.3 Compilação de `mb`

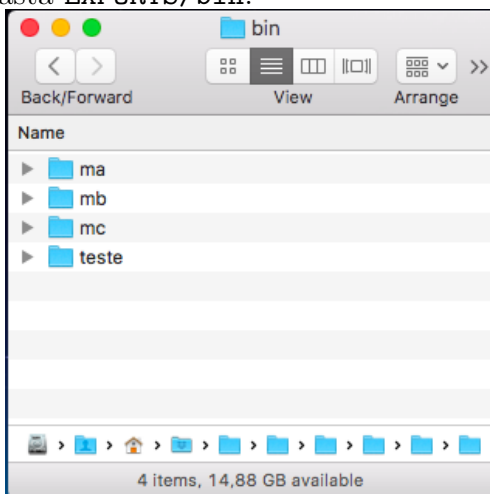
O módulo `mb` agrupa os pacotes `mb.p1b` e `mb.p2b`, que implementam as classes `B1` e `B2`, e as exporta conforme definido no arquivo `module-info.java` do Prog. 12.2. A Fig. 12.6 resume a organização do módulo `mb`, formado pelos pacotes `p1b` e `p2b`.

E as classes definidas pelos pacotes exportados por `mb` são:

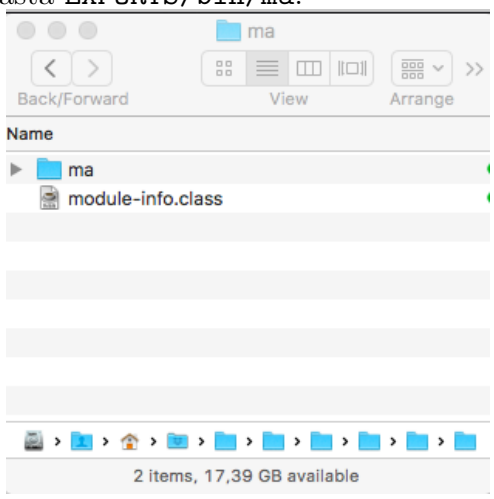
- Arquivo `B1.java` do pacote `mb.p1b`:

```
1  package mb.p1b;
2  public class B1 {
3      public static String name() { return "B1"; }
4  }
```

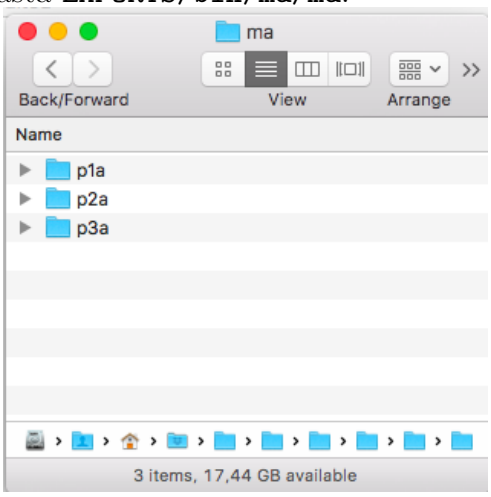
Pasta `EXPORTS/bin`:



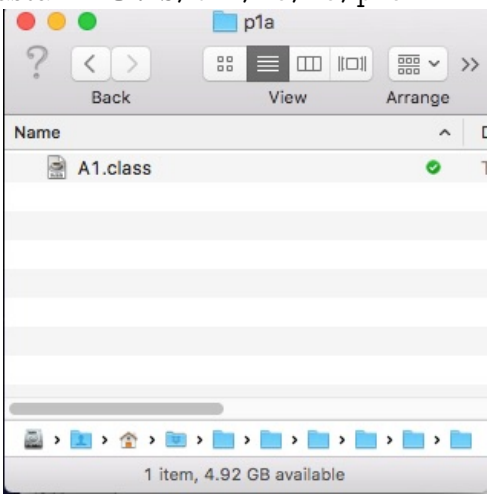
Pasta `EXPORTS/bin/ma`:



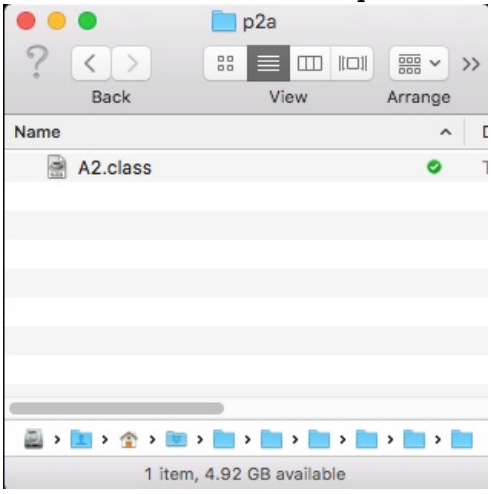
Pasta `EXPORTS/bin/ma/ma`:



Pasta `EXPORTS/bin/ma/ma/p1a`:



Pasta `EXPORTS/bin/ma/ma/p2a`:



Pasta `EXPORTS/bin/ma/ma/p3a`:

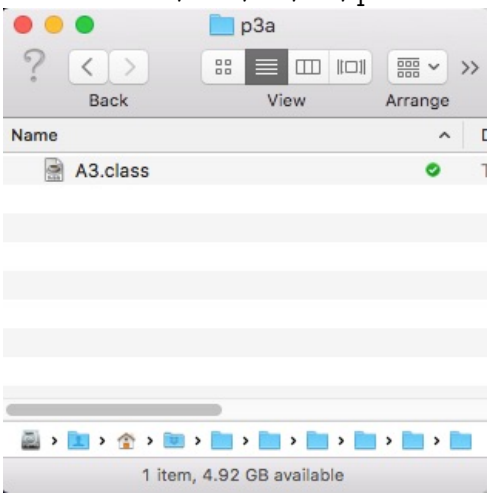


Figura 12.5 Binários de `ma`

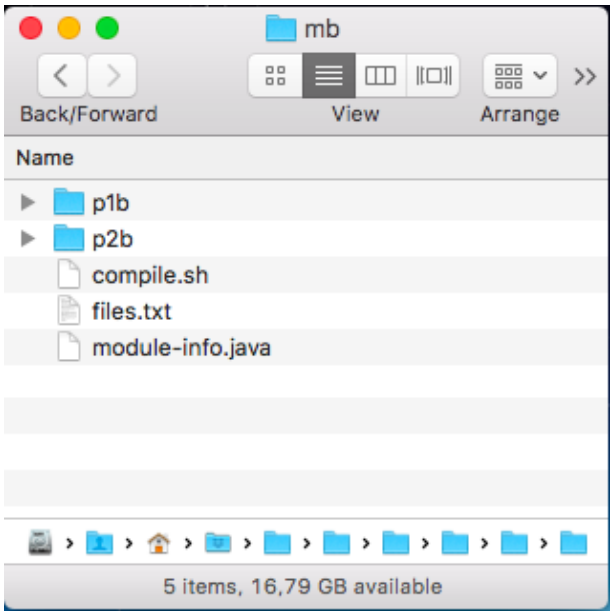


Figura 12.6 Módulo `mb`

- Arquivo `B2.java` do pacote `mb.p2b`:

```
1  package mb.p2b;
2  public class B2 {
3      public static String name() { return "B2"; }
4  }
```

A compilação do módulo `mb` requer um arquivo `files.txt` contendo:

```
module-info.java
p1b/B1.java
p2b/B2.java
```

o qual é usado no comando `javac -d ../../bin/mb @files.txt` do arquivo *shell* `compile.sh` da pasta `mb`.

12.2.4 Compilação `mc`

O módulo `mc`, cuja estrutura é exibida na Fig. 12.7, agrupa os pacotes `mc.p1c` e `mc.p2c`, que implementam as classes `C1` e `C2`,

e as exporta, conforme o arquivo de interface `module-info.java` de `mc`, exibido no Programa 12.2.

Observe que a interface do módulo `mc`, além de exportar os tipos dos pacotes `p1c` e `p2c`, especifica a diretiva `require ma` para ter acesso aos tipos exportados por `ma`.

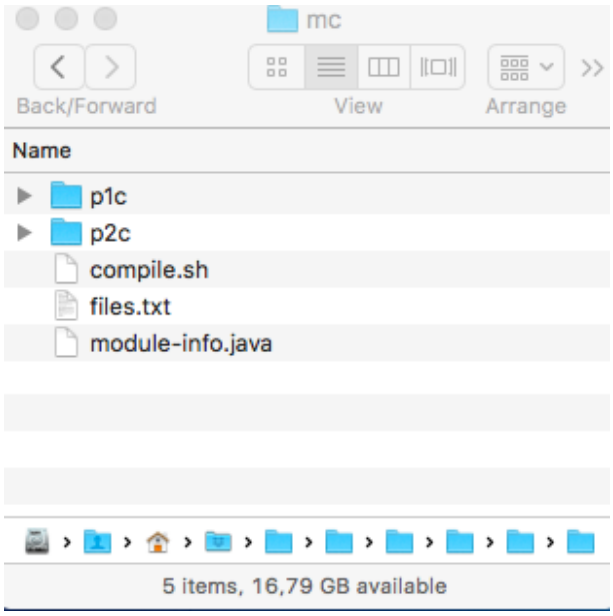


Figura 12.7 Módulo `mc`

Para realizar a compilação, estão na pasta `EXPORTS/src/mc` os arquivos `compile.sh` e `files.txt`. O arquivo *shell* `compile.sh` define o comando para a compilação dos arquivos-fonte do módulo descrito no arquivo `files.txt`, usando os caminhos dos módulos importados e do destino dos `bytecodes` especificados, supondo que o disparo de `javac` seja a partir da pasta `mc`:

```
javac --module-path ../../bin -d ../../bin/mc @files.txt
```

O arquivo `files.txt` especifica os fontes de `mc` a ser compilados e o esquema de visibilidade definido para o módulo:

```
module-info.java
p1c/C1.java
p2c/C2.java
```

E as classes `C1` e `C2` definidas pelos pacotes exportados por `mc` têm o seguinte código:

- Arquivo `C1.java` do pacote `p1c`:

```
1  package mc.p1c;
2  public class C1 {
3     public static String name() { return "C1"; }
4  }
```

- Arquivo `C2.java` do pacote `p2c`:

```
1  package mc.p2c;
2  import  ma.p2a.A2;
3  public class C2 {
4     public static String name() {
5         return "C2 e " + A2.name();
6     }
7  }
```

12.2.5 Compilação de `teste`

A pasta `teste`, que contém o programa principal, armazena os arquivos mostrados na Fig. 12.8, onde destaca-se a pasta `principal`, que contém o arquivo `Main.java`, detalhado no Programa 12.9, e que é a classe principal de `EXPORTS`. E lembre-se que o arquivo `module-info.java` da Fig. 12.8 é o seguinte:

```
1 module teste {
2    requires mb;
3    requires mc;
4 }
```

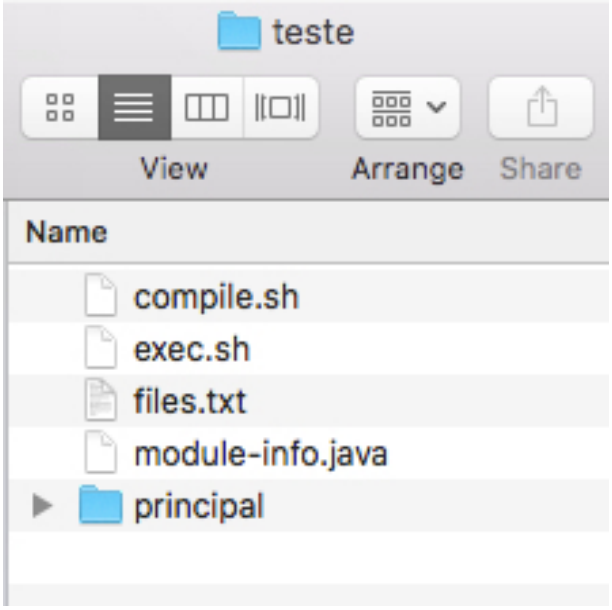


Figura 12.8 Módulo `teste`

```
1  package principal;
2  import mb.p1b.B1;
3  import mb.p2b.B2;
4  import mc.p1c.C1;
5  import mc.p2c.C2;
6
7  public class Main {
8     public static void main(String[] args) {
9        System.out.format("Greetings %s!%n", C1.name());
10       System.out.format("Greetings %s!%n", C2.name());
11       System.out.format("Greetings %s!%n", B1.name());
12       System.out.format("Greetings %s!%n", B2.name());
13    }
14 }
```

Programa **12.9** Classe Main

Observe que as diretivas das linhas 2 a 5 do Programa 12.9, que fazem a importação dos tipos `B1`, `B2`, `C1` e `C2`, são autorizadas pela diretivas **`requires`** do módulo **`teste`**.

O arquivo **`files.txt`** usado na compilação do módulo **`teste`** contém:

```
module-info.java
Main.java
```

O comando de compilação armazenado em `compile.sh`:

```
javac --module-path ../../bin -d ../../bin/teste @files.txt
```

E `exec.sh` com o comando de execução da aplicação:

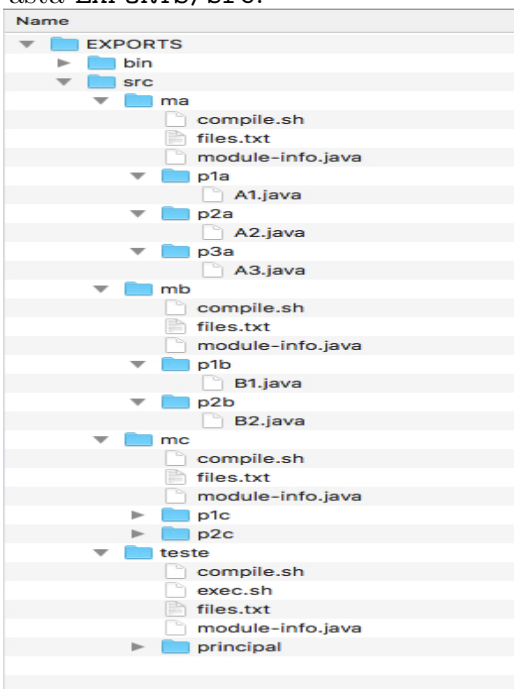
```
java --module-path ../../bin -m teste/teste.Main
```

Arquivo `Main.java` com o fonte do programa `EXPORTS` está definido no pacote `teste`, que contém as importações de tipos de pacotes pertecentes aos módulos `mb` e `mc`.

12.2.6 Cenário final

Para completar o quadro, depois de realizadas as compilações dos módulos `ma`, `mb`, `mc` e `teste`, as pastas que compõem a aplicação `EXPORTS` têm a estrutura hierárquica apresentada na Fig. 12.10.

Pasta `EXPORTS/src`:



Pasta `EXPORTS/bin`:

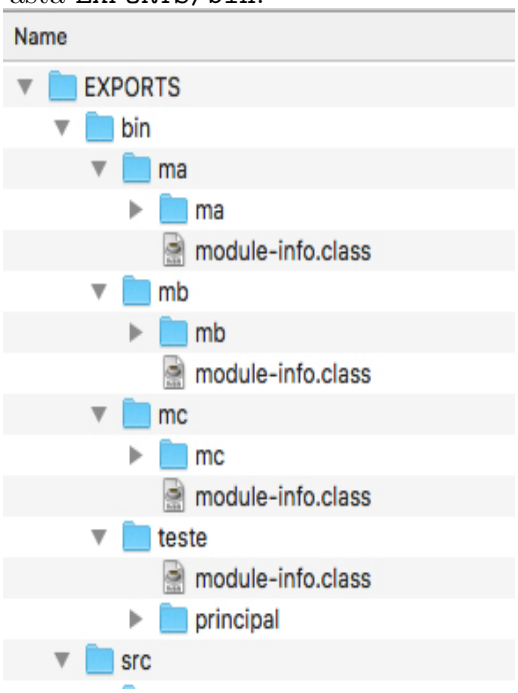


Figura 12.10 Fontes e Binários de `EXPORTS`

Concluindo o exemplo, o disparo da execução `EXPORTS` indicada no módulo `teste`, onde está arquivo `module-info.class`, isto é, na pasta `EXPORTS/bin/teste`, pode ser via o comando:

`java` `--module-path ../../bin` `-m teste/principal.Main`

módulos usados — classe principal

produz o seguinte resultado:

```
Greetings C1!
Greetings C2 e A2!
Greetings B1!
Greetings B2!
```

12.3 Diretivas de módulos

A construção `module` de Java é recurso para organizar a visibilidade de nomes exportados por pacotes, os quais podem ser declarados locais a módulos ou públicos, e a dependência entre módulos.

Todo módulo tem um arquivo de nome `module-info.java`, que define sua interface com outros módulos. Esse arquivo deve ser localizado no diretório raiz das pastas que compõem o módulo.

O arquivo `module-info.java` consiste na especificação de uma lista de diretivas que estabelecem dependência entre módulos e a visibilidade de nomes entre eles. As diretivas definidas são as seguintes:

- `exports` <pacote>
- `exports` <pacote> `to` <módulos>
- `requires` <module>
- `requires transitive` <module>
- `requires static` <module>
- `opens`<pacote>
- `opens`<pacote> `to` <módulos>

- `provides` <tipo> `with` <tipos>
- `uses` <tipo>

12.3.1 Diretiva `exports` <pacote>

A diretiva `exports` de um módulo `mx` que concede aos módulos importadores acessibilidade aos tipos de `mx`, tanto em tempo de compilação como de execução, pode assumir dois formatos:

- `exports p`
- `exports p to p1, p2, ...`

A diretiva `exports p` especificada por um módulo `mx` indica que os tipos públicos do pacote `p` são acessíveis a módulos cuja interface declarar `requires mx`.

A diretiva `exports p to p1, p2, ...` é mais seletiva, pois define a lista de pacotes `p1, p2, ...` que têm permissão de usar o pacote `p` exportado.

12.3.2 Diretiva `requires` <module>

A diretiva `requires` especificada por `mx` define sua dependência de tipos definidos em outros módulos. Essa diretiva pode ter três formatos:

- `requires m`
- `requires transitive m`
- `requires static m`

Em Java, os tipos usados em um programa devem, em princípio, estar presentes tanto durante a compilação quanto na execução. Na compilação, é sempre necessário para realizar verificação de tipos, enquanto na execução sua presença somente é necessária para acesso a seus atributos.

Assim, a diretiva `requires` exige, por default, que todos os tipos necessários estejam presentes na compilação e na execução, sejam eles usados ou não.

Por outro lado, `requires static m` exige a presença do tipos definidos nos pacotes em `m` somente durante compilação. Nesse caso, mensagens de erro serão emitidas somente se algum tipo do pacote requerido forem efetivamente usados não for encontrado na execução.

Se o módulo `mx` declara a diretiva `requires transitive m`, então módulos que se declaram dependentes de `mx` são dependentes do módulo `m`, e todos esses módulos têm acesso aos tipos de `m`.

Para ilustrar o uso de `requires transitive`, suponha uma aplicação denominada `Requires` e que consiste de quatro módulos `ma`, `mb`, `mc` e `md`, organizados conforme a Fig. 12.11.

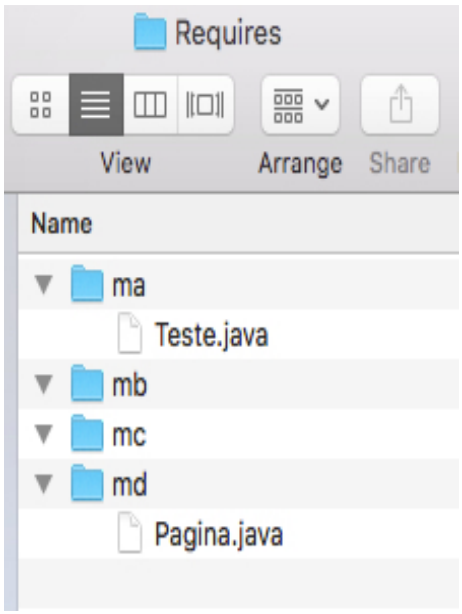


Figura 12.11 Uso de `require`

Suponha o esquema de visibilidade dos módulos `ma`, `mb`, `mc` e `md` seja o definido pelos arquivos `module-info.java` dos módulos da aplicação, conforme detalhado na Fig. 12.12.

`module-info.java` de `ma`:

```
1 module ma {
2   requires mb;
3 }
```

`module-info.java` de `mb`:

```
1 module mb {
2   requires transitive mc;
3 }
```

`module-info.java` de `md`:

```
1 module md {
2   exports texto;
3 }
```

`module-info.java` de `mc`:

```
1 module mc {
2   requires transitive md;
3 }
```

Figura 12.12 Uso de `transitive`

O pacote `texto` exportado por `md` é o do arquivo `Pagina.java`:

```
1 package texto;
2 public class Pagina { }
```

E o arquivo `Teste.java` de `ma` contém o seguinte código:

```
1 package livro;
2 import texto.Pagina;
3 public class Teste {
4    public static void main(String[ ] args) {
5       System.out.prinln(new Pagina( ));
6    }
7 }
```

Observe que, embora `ma` não tenha especificado `requires md`, as referências, em `ma`, ao tipo `p.Pagina` do módulo `md` está correta, porque, no esquema de visibilidade pelos `module-info.java` da Fig. 12.12, está especificado que módulo `mb` `requires transitive mc` e que `mc requires transitive md`.

12.3.3 Diretiva `opens` <pacote>

Módulos podem ser **normais**, se liberam acesso em tempo de compilação e de execução somente aos pacotes explicitamente ex-

portados, ou **abertos**, se liberam acesso em tempo de compilação aos pacotes explicitamente exportados, mas permitem acesso em tempo de execução, via reflexão. A diretiva `opens <pacote>` promove a abertura do acesso a todos os módulos, como no seguinte exemplo.

```
1 module ma {
2     exports ma.p1a;
3     exports ma.p2a;
4     opens ma.p3a;
5 }
```

onde os pacotes `p1a`, `p2a` e `p3a` são os definidos no módulo `ma`, e que permitem a escrita do seguinte trecho de programa em algum pacote fora do módulo `ma`:

```
1 ...
2 Class c = Class.forName("ma.p3a.A3");
3 Method m = c.getMethod("name");
4 System.out.println("-----" + m.invoke(...));
5 ...
```

A diretiva `opens <pacote> to <modules>` promove a abertura somente para módulos específicos.

12.3.4 Diretiva `provides` <tipo> `with` <tipos>

A diretiva `provides` serve para indicar que um módulo provê um serviço de implementação, sendo, portanto, um provedor de serviços.

Após a palavra-chave `provides` deve-se listar o nome da interface ou da classe abstrata do serviço, e após `with`, a classe que a implementa. No exemplo abaixo, o módulo `Serviço` define o pacote que contém o tipo do serviço que deve ser fornecido pelo módulo `Provedor` ao módulo `Consumidor`.

```
module Provedor {
    requires Serviço;
    provides javax0.serviceinterface.ServiceInterface
      with javax0.serviceprovider.Provider;
}
module Consumidor {
    requires Serviço;
    uses javax0.serviceinterface.ServiceInterface;
}
module Serviço {
    exports javax0.serviceinterface;
}
```

12.3.5 Diretiva `uses` <tipo>

A diretiva **`uses`** declara o tipo dos serviços que o módulo consume. Esses serviços podem ser a implementação de uma classe, interface ou anotação para ser usados por outras classes.

A diretiva **`requires`** também provê um serviço a ser consumido por um dado módulo, mas se esse serviço implementa uma interface definida por módulo transitivamente dependente, tem-se que especificar essa transitividae para alcançá-lo.

A diretiva **`uses`** simplifica esse processo, provendo diretamente a interface desejada no caminho de acesso ao módulo desejado.

12.4 Conclusão

A construção **`module`** juntamente com classes e pacotes formam um conjunto de poderosos mecanismos de Java para controlar visibilidade de nomes em um programa modular.

Sabendo usar, esses mecanismos permitem colocar em prática a mensagem do verso "*Good fences make good neighbors*" do Poema

Mending Wall de Robert Frost, no sentido de atingir alto grau de modularização.

Exercícios

1. Qual é a diferença entre o conceito de módulo da Teoria de Programação Modular e a construção `module` de Java?
2. Avalie o uso da estrutura do sistema de arquivos do ambiente de programação como um recurso para controlar visibilidade de tipos entre pacotes. Há alguma alternativa mais interessante?
3. Em que momento, durante a execução de um programa, as classes nele mencionadas são alocadas na memória?
4. Analise o verso "*Good fences make good neighbors*" do Poema *Mending Wall* de Robert Frost do ponto de vista de programação modular.

Notas bibliográficas

Os textos mais importantes para estudar o uso de `module` para controle de visibilidade entre pacotes são a página de Java mantida pela Oracle (`http://www.oracle.com`) e o documento de especificação de Java SE 14 de James Gosling et alii [22].

Capítulo 13

Genéricos

Na linguagem Java, todo objeto é uma instância direta ou indireta da classe `Object`, que é a raiz de todas hierarquias de classes. Assim, onde um objeto do tipo `Object` for esperado, pode-se usar um outro de qualquer tipo. Essa facilidade permite o desenvolvimento de classes que lidam com objetos de tipos arbitrários, sendo assim dotadas de um grau mais elevado de reúso.

Com frequência trabalha-se com classes do tipo contêiner, que são assim chamadas porque encapsulam conjuntos de objetos de um dado tipo. As operações dessas classes funcionam de maneira uniforme independentemente do tipo dos objetos por elas encapsulados.

Há também o caso de funções que produzem resultados independentemente dos tipos de seus argumentos. Por exemplo, uma função de ordenação recebe como parâmetro um vetor de certos valores e devolve o mesmo vetor com seus valores ordenados, não importando o tipo dos seus elementos. Exige-se apenas que eles sejam comparáveis entre si, conforme o critério de ordenação.

Nesses casos, ganha-se generalidade ao trabalhar com o tipo `Object` em vez do tipo mais específico. Por exemplo, uma pilha de `Integer` apresenta rigorosamente o mesmo comportamento de uma pilha de `String`. Assim, não se deveria ter duas implementações para dar conta dessas duas pilhas. Basta que se defina

uma classe que implemente uma pilha de `Object` e que se passem os parâmetros das operações de empilhar e desempilhar adequadamente. Similarmente à função de ordenação, citada acima, pode-se implementar a estrutura como vetor de `Object`, e o mecanismo de polimorfismo de inclusão garante o funcionamento desejado.

Entretanto, o conceito de referências genéricas oferecido por esse tipo de polimorfismo transfere ao programador a responsabilidade de garantir o correto uso dos objetos manipulados. O compilador não tem meios de verificar o uso correto dos tipos. Por exemplo, em uma pilha de `Object`, pode-se empilhar tanto um objeto `String`, quanto um da classe `Integer`. No momento do desempilhamento, o programador deve explicitamente verificar o tipo dos objetos retornados de forma a dar-lhe o devido processamento no código subsequente.

Para resolver essas dificuldades e dar ao compilador os meios de detecção de possíveis erros de mistura indevida de tipos, Java oferece os mecanismos de criação e uso de métodos, classes e interfaces genéricos.

13.1 Métodos genéricos

Frequentemente, há métodos que diferenciam-se apenas no tipo de objetos manipulados e têm corpos exatamente com a mesma estrutura.

Por exemplo, suponha que se deseja ter uma função polimórfica de sobrecarga `imprimaArranjo` com as seguintes assinaturas:

```
void imprimaArranjo(Integer[] a)
void imprimaArranjo(Double[] a)
void imprimaArranjo(Character[] a)
```

e que tenha o objetivo de imprimir os conteúdos de vetores que lhe

forem passados como parâmetros.

Uma primeira solução seria escrever três funções:

```
1 public static void imprimaArranjo(Integer[] a) {
2    for (Integer e: a) System.out.printf("%s ",e);
3 }
4 public static void imprimaArranjo(Double[] a) {
5    for (Double e: a) System.out.printf("%s ",e);
6 }
7 public static void imprimaArranjo(Character[] a) {
8    for (Character e: a) System.out.printf("%s ",e);
9 }
```

A função **main** abaixo ilustra o uso dessas três funções:

```
1 public static void main(String[] args) {
2    Integer[] x = {1,2,3,4,5,6,7,8,9};
3    double[]  z = {1.1, 2.2, 3.3, 4.4};
4    Character [] y = {'O', 'L', 'Á'};
5    imprimaArranjo(x); System.out.println();
6    imprimaArranjo(y); System.out.println();
7    imprimaArranjo(z);
8 }
```

Por outro lado, essas três implementações de **imprimaArranjo** fazem exatamente o mesmo tipo de ação, que independem do tipo dos valores passados como parâmetro. Isso permite, via o recurso de métodos genéricos de Java, trocar as três definições de **imprimaArranjo** acima por um único método:

```
1 public static <E> void imprimaArranjo(E[] a) {
2    for (E e: a) System.out.printf("%s ", e);
3 }
```

Nessa declaração, **E** é uma variável de tipo, que é usada para designar o tipo do parâmetro da função **imprimaArranjo**.

As variáveis de tipo devem ser sempre anunciadas no cabeçalho do método que as usa com um texto da forma `<`E_1`,`E_2`,...>`, colocado imediatamente antes do tipo de retorno do método, onde E_i, para $i \geq 1$, são identificadores Java e servem para declarar parâmetros e variáveis locais no corpo do método. Cada variável de tipo será instanciada conforme o tipo do respectivo argumento de chamada. Por exemplo, no caso de `imprimaArranjo(x)`, `E` será considerado um `Integer`, pois `x` é um vetor de `Integer`, e para `imprimaArranjo(z)`, `Double`.

Note, entretanto, que `E` pode ser substituído por qualquer tipo descendente da classe `Object` e, no corpo de `imprimaArranjo`, somente as operações definidas na classe `Object` podem ser aplicadas objetos do tipo `E`.

13.2 Classes genéricas

Uma classe genérica é uma classe com parâmetros formais que são tipos, os quais são também chamados de variáveis de tipo. Esses parâmetros do tipo tipo são designados por identificadores listados entre os pares de colchetes angulares (`<` e `>`), colocados após o nome da classe, como `T`, `K` e `E` em `class A<T,K,E>{...}`.

Quaisquer identificadores podem ser usados para designar parâmetros de tipo, mas, por convenção, adotam-se identificadores de uma única letra maiúscula, como `E`, `K`, `T` e `V`.

Esses tipos parâmetros formais podem, no momento da especificação de um novo tipo a partir da classe genérica, ser substituídos por qualquer tipo da hierarquia `Object`. Essa operação, chamada de *invocação* ou *instanciação* da classe genérica cria um novo tipo de dados. E o tipo criado é denominado *tipo parametrizado*.

A invocação de uma classe genérica com uma lista de argumentos

de tipo cria um novo tipo que tem validade apenas em tempo de compilação, ou seja, é criado um tipo estático. A compilação da classe genérica cria um tipo que é válido também durante a execução, portanto um tipo dinâmico.

O processo de compilação de uma classe genérica usa uma técnica denominada *apagamento de tipo* (*type erasure*), que remove do código produzido toda informação relativa a tipos parâmetros e tipos argumentos. Por exemplo, os tipos `ArrayList<Integer>` e `ArrayList<String>` são diferentes tipos estáticos, que são usados para a verificação de tipo e depois traduzidos para o tipo `ArrayList`, isto é, ambos são representados pelo mesmo tipo dinâmico, embora sejam tipos distintos durante a compilação. A rigor, a invocação da classe genérica não cria uma nova classe, pois todas as suas *invocações* têm exatamente o mesmo descritor, portanto tratam-se da mesma classe.

A expressão `A<Integer>` designa um novo tipo, resultado da invocação da classe `A` com o parâmetro `Integer`, tal que as ocorrências do tipo `T` no corpo de `A` são interpretadas como `Integer`: o método `s` da classe `A<T>`, para o caso de `A<Integer>`, espera um argumento do tipo `Integer`, e o método irmão `g` retorna um objeto do tipo `Integer`.

Toda classe genérica é compilada para gerar seu descritor, e a cada invocação cria-se um novo tipo baseado no mesmo descritor, compartilhando as mesmas operações. Os tipos criados são distintos, porém compartilham o mesmo descritor de classe e as mesmas operações. Isto evita duplicação de código, uma vez que o descritor de classes contém o código das funções definidas na classe.

Esses detalhes da semântica da invocação de classes genéricas são exibidos no exemplo a seguir, o qual imprime "`descritor único`", para os dois objetos declarados com *invocações* distintas de `A`.

```
1  class A<T> {
2    ... public void s(T a) { ... }
3    ... public T g() { ... }
4  }
5  public class Genérico1 {
6     public static void main(String[] args) {
7        A<Integer> a = new A<Integer>();
8        A<Double>  b = new A<Double>();
9        String t;
10       if (a.getClass() == b.getClass()) t = "único";
11       else t = "distinto!";
12       System.out.print("descritor " + t);
13    }
14 }
```

O programa a seguir ilustra a criação de dois novos tipos a partir classe genérica `A`, a qual é invocada duas vezes, linhas 3 e 7, para construir os tipos `A<Integer>` e `A<Double>`, respectivamente.

```
1  public class Genérico2 {
2     public static void main(String[] args) {
3        A<Integer> m = new A<Integer>();
4        int k, i = 10;
5        m.s(i); k = m.g();
6        System.out.print("Valor de i = " + k + ", ");
7        A<Double> q = new A<Double>() ;
8        double w, t = 100.0;
9        q.s(t); w = q.g();
10       System.out.print("Valor de t = " + w);
11    }
12 }
```

que imprime `Valor de i = 10, Valor de t = 100.0`.

Esses tipos, `A<Integer>` e `A<Double>`, são distintos e incompatíveis, não tendo qualquer relação entre si, exceto que ambos são

instâncias de um mesmo tipo base, e, portanto, subtipos desse tipo base e têm o mesmo descritor de classe.

O programa **Genérico3** a seguir tem erro de tipo, e se submetido ao compilador Java receberá a mensagem

incompatible type in q.s(t) and q.g().

Isso ocorre porque o método **q.s** espera receber como parâmetro um objeto do tipo **double**, e o parâmetro **t**, sendo do tipo **int** somente pode ser empacotado em um objeto **Integer**. Situação similar ocorre com **w = q.g()**, pois não se converte automaticamente **Double** em **int**.

```
1  public class Genérico3 {
2     public static void main(String[] args) {
3        A<Integer> m = new A<Integer>();
4        A<Double>  q = new A<Double>() ;
5        int k, i = 10, w, t = 100;
6        m.s(i);
7        k = m.g();
8        q.s(t);
9        w = q.g();
10    }
11 }
```

No corpo de classe genérica, variáveis de tipo podem ser usadas para declarar atributos da classe, variáveis locais a funções, parâmetros de métodos, tipos de retorno de métodos e parâmetros de invocações de classes genéricas. Funções de classe genérica que usam variáveis de tipo são polivalentes, pois aplicam-se a todas as invocações da classe genérica.

Como o tipo parâmetro deve ser do tipo referência, as operações a ele aplicáveis são somente as herdadas da classe **Object**, incluindo atribuição de referências (=) e comparação de referências por igual (==) e diferente (!=). Nenhuma outra operação é aplicável a objetos declarados com variável de tipo.

13.2.1 Criação de objetos genéricos

Objetos do tipo *parâmetro de tipo* podem ser criados, embora, para esse fim, exija-se uma pequena variação na sintaxe do operador `new`. Por restrições do compilador Java, uma variável de tipo de classe genérica não pode ser operando de `new`, i.e., a construção

```
T x = new T()
```

onde `T` é um tipo parâmetro, é rejeitada pelo compilador. Para se obter o efeito desejado de criar um objeto do tipo designado por `T`, deve-se escrever:

```
T x = (T) new Object()
```

que aloca corretamente o objeto desejado.

Essa diferença na sintaxe da operação de criação de objetos sugere que o código gerado em cada caso seja específico e particular.

A declaração abaixo mostra uma definição da classe genérica `A<T>`, onde `T` especifica seu tipo parâmetro:

```
1 class A<T> {
2    private T x ;
3    public A() {x = (T) new Object();}
4    public void s(T y) {x = y;}
5    public T g() {return x;}
6 }
```

O parâmetro formal ou variável de tipo `T` é usado na classe `A` para declarar o atributo `x`, o parâmetro `y` do método `s` e o tipo do valor de retorno de `g`.

13.3 Tipos primitivos e parâmetros

Para facilitar a escrita de programas, valores de tipos básicos podem ser usados onde o correspondente tipo invólucro for esperado,

sendo o empacotamento e desempacotamento de valores básicos realizados automaticamente.

Entretanto, os tipos que substituem o parâmetro de tipo da classe genérica no momento da invocação da classe não podem ser tipos básicos. Caso deseje-se instanciar uma classe genérica com tipos básicos, as correspondentes classes-invólucro (*wrapped class*), como `Integer`, associada ao tipo `int`, devem ser explicitamente usadas no lugar dos tipos primitivos.

Por exemplo, o programa abaixo, que mostra empacotamento e desempacotamento automáticos do tipo básico `int` na linha 5, imprime `Valor de i = 100`.

```
1 public class Genérico4 {
2    public static void main(String[] args) {
3       A<Integer> m = new A<Integer>();
4       int k, i = 100;
5       m.s(i); k = m.g();
6       System.out.print("Valor de i = " +  k);
7    }
8 }
```

Especificamente, comando `m.s(i)` do programa `Genérico4` somente é válido porque valores de tipos primitivos, e.g., `int`, são automaticamente *empacotados* em um objeto da classe *wrapped* correspondente, no caso, o tipo `Integer`, sempre que o contexto assim exigir. O que é passado ao método `s` é a referência ao objeto do tipo `Integer` criado a partir do valor de `i`.

Situação análoga ocorre com a atribuição `k = m.g()`, onde o valor da referência a objetos do tipo `Integer` retornado por `m.g()` é automaticamente *desempacotado* para obter o valor `int` a ser atribuído ao inteiro `k`.

13.4 Tipos brutos

É possível utilizar uma classe genérica para declarar objetos sem especificar seus argumentos de tipo. Nesse caso, é criado uma instância da classe genérica usando implicitamente o tipo `Object` com argumento. Os objetos assim declarados têm tipos brutos (*raw types*). Por exemplo, dada a classe genérica `A<T>`, escrever

```
A a = new A()
```

é equivalente a

```
A<Object> a = new A<Object>().
```

A referências a objetos de tipo bruto pode-se atribuir referências a objetos de classes instanciadas da classe genérica, como em

```
A a = new A<Integer>().
```

A atribuição invertida, `A<Integer> b = a`, é permitida pelo compilador, mas deve ser evitada, pois é perigosa, uma vez que não se pode garantir que os objetos apontados por `a` armazenam sempre objetos `Integer`.

13.5 Hierarquia de genéricos

Classes genéricas ou suas instâncias podem ser estendidas para formar novas classes, que podem ser genéricas ou não. Por exemplo, a declaração

```
class B extends A<Double> {...}
```

cria uma nova classe não-genérica de nome `B` a partir da instância `A<Double>` da classe genérica `A<T>`.

E o polimorfismo de inclusão de Java continua válido nesse caso, pois referências a objetos do tipo `B` podem ser usadas quando esperam-se referências a objetos do tipo `A<T>` ou `A<Double>`.

Para a invocação de uma classe genérica, os parâmetros fornecidos devem ser tipos válidos no escopo.

O programa `Genérico5` abaixo produz o mesmo resultado que `Genérico2` apresentado acima. A única diferença entre eles é o uso, na linha 7, do comando

```
B q = new B()
```

em vez de

```
A<Double> q = new A<Double>().
```

```
1  public class Genérico5 {
2     public static void main(String[] a) {
3        A<Integer> m = new A<Integer>();
4        int k,i = 10;
5        m.s(i); k = m.g();
6        System.out.print("Valor de i = " + k + ", ");
7        B q = new B();
8        double w, t = 100.0;
9        q.s(t);  w = q.g();
10       System.out.print("Valor de t = " + w);
11    }
12 }
```

Quando uma classe genérica é estendida ou especializada, seus parâmetros devem ser supridos adequadamente. Caso a subclasse desejada seja genérica, suas variáveis de tipo podem ser usadas para instanciar a superclasse genérica, com mostram os seguintes exemplos de especialização:

```
1 class C<H> extends A<H> { }
2 class D<T,R> extends A<T> {
3      private R z;
4      public D(R v) {super(); z = v;}
5      public R g() {return z;}
6 }
7 class E extends A<Integer> { }
```

Note que os parâmetros da classe genérica derivada, e.g., `H` de `C`, `T` e `R` de `D`, são usados para definir a instância das superclasses

especificadas.

O programa **Genérico6**, que ilustra o uso das classes genéricas `A` e `D`, imprime `i = 10, b = 100, c = 1000.0`.

```
1  public class Genérico6 {
2     public static void main(String[] args) {
3        A<Integer> m = new A<Integer>();
4        int k,i = 10;
5        m.s(i); k = m.g();
6        System.out.print("i = " + k + ", ");
7        double c = 1000.0;
8        D<Integer,Double> q = new D<Integer,Double>(c);
9        int a, b = 100;
10       double t;
11       q.s(b); a = q.g();
12       System.out.print("b = " + a + ", ");
13       t = q.h();
14       System.out.print("c = " + t);
15    }
16 }
```

Por garantir a compilação de programas de versões de Java mais antigas pelos novos compiladores de Java, permite-se estender classes genéricas sem especificação de seus argumentos, como em:

```
class B extends A
```

que tem a interpretação de

```
class B extends A<Object>
```

Essa convenção permite que programas escritos em versões de Java anteriores a 5.0 e que usam, por exemplo, a classe `LinkedList`, continuem a funcionar em ambientes Java de versão 5.0 ou maior, nos quais as classes da família `Collection` foram substituídas por versões parametrizadas, como `LinkedList<T>`.

Por outro lado, a declaração `class F extends A<T> { }` estará sintaticamente errada se `T` estiver indefinido nesse ponto.

13.6 Interfaces genéricas

Interfaces genéricas são interfaces com um ou mais parâmetros que são tipos e que podem ser usados para especificar a assinatura dos elementos da interface, como mostra a seguinte declaração:

```
1 public interface I<T> {
2    void s(T y);
3    T g ();
4 }
```

Interfaces parametrizadas podem ser normalmente estendidas para formar hierarquias de interfaces genéricas e também ser implementadas por meio de classes não-genéricas ou genéricas.

Um exemplo do uso de interface é ilustrado pela seguinte implementação da classe genérica `A<T>` e da classe não-genérica `B`.

```
1 public class A<T> implements I<T> {
2    private T x ;
3    public A() { }
4    public void s(T y) {x = y;}
5    public T g() { return x;}
6 }
7 public class B implements I<Double> {  }
```

O programa `Genérico7` imprime `t = 10`.

```
1 public class Genérico7 {
2    public static void main(String[] args) {
3       A<Integer> q = new A<Integer>();
4       int w, t  = 10;
5       q.s(t);
6       w = (Integer)q.g();
7       System.out.print("t = " + w);
8    }
9 }
```

13.7 Limite superior de tipo

Variáveis de tipo ou parâmetros de classes genéricas podem corresponder a quaisquer referências de objetos pertencentes a classes da hierarquia baseada em `Object`. Em consequência, somente as operações aplicáveis a `Objecct` são garantidamente válidas em todo uso dos parâmetros da classe genérica. O uso de qualquer outra operação na manipulação de objetos cujo tipo seja uma variável de tipo é rejeitada pelo compilador.

Entretanto, há situações em que se deseja restringir os parâmetros de uma classe genérica a hierarquias específicas, permitindo que operações de uma dada classe, além daquelas definidas por `Object`, possam ser também aplicadas a referências do tipo parâmetro da classe genérica.

O recurso de Java para esse fim denomina-se *Parâmetro com Limite Superior*, que permite vincular o parâmetro a uma hierarquia de tipos. Para isso, no lugar de um parâmetro de tipo `<T>` deve-se usar `<T extends R>` para vincular `T` à hierarquia de `R`.

No exemplo a seguir, dentro da classe `M`, as operações aplicáveis a objetos do tipo `T` são as definidas para objetos da classe `R`, no caso, o método `f`, na linha 3, e os herdados de `Object`.

```
1 public class M<T extends R> {
2     private int z;
3     public void s(T y) {z = y.f();}
4     public int  g() {return z;}
5 }
6
7 public class R {
8    public int f() {return 1;}
9 }
```

Usuários das classes `M<T>` poderiam declarar as classes `R1` e `R2` abaixo e então usar objetos com os tipos `M<R1>` e `M<R2>`, que teriam garantidamente a operação `f` declarada em `R`.

```
1  public class R1 extends R {
2     public int g() {return 100;}
3  }
4
5  public class R2 extends R {
6     public int f() {return 1000;}
7  }
```

O limite superior definido para o parâmetro, que no exemplo acima é a classe `R`, poderia ser uma interface, conforme é ilustrado pela seguinte implementação alternativa da classe `M`:

```
1  class M<T extends R> {
2      private z;
3      public void s(T y) { z = y.f() ;}
4      public int  g() { return z;}
5  }
6
7  interface R { public int f() }
8
9  class R1 implements R {
10    public int f() {return 1;}
11    public int g() {return 100;}
12 }
13
14 class R2 implements R {
15    public int f() {return 1000;}
16 }
```

O programa `Genérico8`, onde a classe `M` usada pode ser qualquer uma das implementações acima, imprime

`k = 100, j = 1000`:

```
1  public class Genérico8 {
2     public static void main(String[] args) {
3        M<R1> m = new M<R1>();
4        M<R2> r = new M<R2>();
5        R1 a = new R1();
6        R2 b = new R2();
7        int j, k;
8        m.s(a); r.s(b);
9        k = m.g();  j = r.g();
10       System.out.print("k = "+ k + ", j = "+ j);
11    }
12 }
```

Tentativas de instanciar a classe `M` definida acima com tipos fora da família de `R` serão apontadas como erro de compilação, como ocorre no programa abaixo:

```
1  public class Genérico9 {
2     public static void main(String[] args) {
3        M<Integer> m = new M<Integer>(); // ERRO
4        int j, k = 1;
5        m.s(k);  j = r.g();
6        System.out.print("k = "+ k + ", j = " +  j);
7     }
8  }
```

Na definição de tipos vinculados, a palavra-chave **`extends`** pode significar tanto **`extends`** *classe* como **`implements`** *interface*, dependendo do nome que segue a palavra **`extends`**.

No caso de se desejar vincular o tipo parâmetro a uma classe e a várias interfaces simultaneamente, deve-se declarar que o parâmetro tipo **`extends`** uma lista de tipos, separados por `&`, sendo o primeiro da lista o nome de uma classe ou de uma interface, e os demais somente nomes de interfaces. Por exemplo, as declarações a seguir

```
1 class A<E extends C & I1 & I2 & I3> { ... }
2 class C { ... }
3 interface I1 { ... }
4 interface I2 { ... }
5 interface I3 { ... }
```

requerem que a classe genérica `A` seja invocada com parâmetros que estendam a classe `C` e implementam as interfaces `I1`, `I2` e `I3`.

O conceito de limite superior de tipos também se aplica a métodos genéricos. Por exemplo, deseja-se definir um método `max(a,b,c)` para retornar o maior valor dentre `a`, `b` e `c`, os quais podem ser de qualquer tipo que tenha a operação `compareTo`. O primeiro passo consiste na especificação de uma interface genérica para definir o tipo limite contendo a assinatura do método `compareTo`:

```
interface Comparable<T> {boolean compareTo(T);}
```

a qual é uma interface que faz parte da biblioteca de Java. O método genérico `max` pode então ser definido da forma abaixo.

```
1  class Comparadores {
2     public static <T extends Comparable<T>> T
3                                  max(T x, T y, T z) {
4        T m = x;
5        if (y.compareTo(m) > 0)  m = y;
6        if (z.compareTo(m) > 0)  m = z;
7        return m;
8     }
9    public static void main(String[] args) {
10      System.out.print(max(3, 4, 5));
11      System.out.print(max(3.3, 5.0, 4.0));
12      System.out.print
13        (max("W. Mozart", "L. Beethoven", "A. Vivaldi"));
14   }
```

Os tipos dos argumentos de `max` podem ser novos tipos definidos pelo programador ou tipos já existentes. Exige-se apenas que sejam

da hierarquia de `Comparable`, como `Integer`, `Double` e `String`. A função `main` da classe `Comparadores` acima, que ilustra usos de `max`, imprime os valores `5`, `5.0` e `W. Mozart`.

A invocação de uma classe genérica para instanciar um novo tipo não cria cópia de seu descritor, como mostra o seguinte código:

```
A<String>  x = new A<String>("um texto");
A<Integer> y = new A<Integer>(10);
System.out.print("Objetos x e y ");
if (x.getClass()!= y.getClass())System.out.print("NÃO");
System.out.print(" têm o mesmo descritor.");
```

que imprime o texto `Objetos x e y têm o mesmo descritor.`

13.8 Tipo curinga

As classes `Integer` e `Double` são subclasses de `Number`, assim pode-se misturar valores desses tipos em um arranjo de `Number`:

```
import java.util.ArrayList;
public class Somador1 {
   public static void main(String[] args) {
      Number[] numbers = {1, 2.0, 3, 4.0, 5};
      ArrayList<Number> list = new ArrayList<Number>();
      for (Number e: numbers) list.add(e);
      System.out.println("Números: " + list);
      System.out.println("Soma: " + sum(list));
   }
   public static double sum(ArrayList<Number> list) {
      double t = 0.0;
      for(Number e : list) t += e.doubleValue();
      return t;
   }
}
```

A execução do programa `Somador1` produz:

```
Números: [1, 2.0, 3, 4.0, 5]
Soma: 15.0
```

Por outro lado, se `A<T>` for uma classe genérica, `A<Number>` não quarda relação alguma com `A<Integer>`, exceto que tenham, ambos, o mesmo descritor de classe, i.e., o descritor de `A`.

Para ilustrar o fato que instâncias de tipos genéricos são tipos distintos, considere a implementação abaixo de um método `sum` para somar uma lista de valores numéricos, do tipo `int` ou `double`, que estão armazenados em objetos do tipo `ArrayList<E>` da biblioteca `java.util`, e que deseja-se somar numéricos do tipo `int` e do tipo `double` usando o mesmo método `sum`.

A solução *óbvia*, embora equivocada, seria criar as listas `list1` e `list2` de `Integer` e de `Double`, conforme definidas abaixo:

```
1  import java.util.ArrayList;
2  public class Somador2 {
3     public void static main(String[] args) {
4        Integer [] integers = {1, 2, 3, 4, 5}
5        Double []  doubles  = {1.0, 2.0, 3.0, 4.0, 5.0}
6        Arraylist<Integer> list1 = new Arraylist<Integer>();
7        for (Number e: integers) list1.add(e);
8        System.out.println("Inteiros: " + list1);
9        System.out.println1h("Soma inteiros: " + sum(list1));
10       Arraylist<double> list2 = new Arraylist<Double>();
11       for (Number e: doubles) list2.add(e);
12       System.out.println("Duplos: " + list2);
13       System.out.println("Soma duplos: " + sum(list2));
14    }
15    public static double sum(Arraylist<Number> list) {
16       double t = 0.0;
17       for(Number e : list) t += e.doubleValue();
18       return t;
19    }
20 }
```

Entretanto, a compilação de **Somador2** produz mensagens de erro relativas às linhas 6, 9, 10 e 13.

13.8.1 Curinga com limite superior de tipo

O programa **Somador3** resolve o problema apontado em **Somador2** redeclarando o cabeçalho de **sum**, da linha 16 com um parâmetro de tipo curinga:

```
1  import java.util.ArrayList;
2  public class Somador3 {
3     public static void main(String[] args) {
4        Integer [] integers = {1, 2, 3, 4, 5};
5        Double []  doubles  = {1.0, 2.0, 3.0, 4.0, 5.0};
6        ArrayList<Integer> list1=new ArrayList<Integer>();
7        for (Integer i : integers) list1.add(i);
8        System.out.println("Inteiros: " + list1);
9        System.out.println("Soma inteiros: " + sum(list1));
10       ArrayList<Double> list2 = new ArrayList<Double>();
11       for (Double d : doubles) list2.add(d);
12       System.out.println("Duplos: " + list2);
13       System.out.println("Soma duplos: " + sum(list2));
14    }
15    public static double sum(
16                   ArrayList<? extends Number> list) {
17       double t = 0.0;
18       for(Number e : list) t += e.doubleValue();
19       return t;
20    }
21 }
```

`ArrayList<Integer>` e `ArrayList<Double>` são tratados como subtipos de `Arraylist<? extends Number>`, graças ao uso do tipo curinga e ao fato de **Integer** e **Double** serem subclasses de **Number**.

A execução do programa **Somador3** produz o seguinte resultado:

```
Inteiros: [1, 2, 3, 4, 5]
Soma inteiros: 15.0
Duplos: [1.0, 2.0, 3.0, 4.0, 5.0]
Soma duplos: 15.0
```

Como curingas representam um tipo desconhecido, não se pode fazer qualquer operação que exija conhecimento do tipo.

13.8.2 Curinga com limite inferior

A especificação de tipo `T<? extends R>` limita o parâmetro de `T` aos membros descendentes da hierarquia de `R`. É também possível a especificação `T<? super R>`, que restringe os parâmetros de `T` a tipos que sejam membros ascendentes de `R` em sua hierarquia. Um método, como o `print` de `Super1` abaixo, declarado para receber um argumento do tipo `ArrayList<? super Integer>` aceita argumentos do tipo `ArrayList<Integer>`, `ArrayList<Number>` ou `ArrayList<Object>`.

```
1  import java.util.ArrayList;
2  public class Super1 {
3     public static void main(String[] args) {
4        Integer [] integers = 1, 2, 3, 4, 5;
5        ArrayList<Number> list1 = new ArrayList<Number>();
6        for (Integer i : integers) list1.add(i);
7        System.out.println("Inteiros: " + list1);
8        print(list1);
9     }
10    public static void print(
11                      ArrayList<? super Integer> list) {
12       for(Object e : list) System.out.print(" " + e);
13       System.out.println();
14       return;
15    }
16 }
```

O programa **Super1** imprime

```
Inteiros: [1, 2, 3, 4, 5]
 1 2 3 4 5
```

É preciso tomar cuidado no uso do parâmetro do tipo curinga com limite inferior, pois deve-se sempre supor que o objeto passado use o tipo parâmetro mais alto na hierarquia, i.e., **Object**, como feito no programa acima. Por exemplo, se a função **print** do programa **Super1** for substituída por:

```
1    public static void print(
2                    ArrayList<? super Integer> list) {
3       for(Integer e : list) System.out.print(" " + e);
4       System.out.println();
5       return;
6    }
```

o compilador Java emitirá mensagem informando que os elementos de **list** no comando

```
for(Integer e : list) System.out.print(" " + e);
```

não pode sempre ser convertido a **Integer**.

13.9 Conclusão

Genéricos são um recurso para construção de código de elevado grau de reúso, que é demandado quando trabalha-se com classes do tipo contêiner, que encapsulam conjuntos de objetos de um dado tipo, e cujas operações funcionam de maneira uniforme independentemente do tipo dos objetos por elas encapsulados.

Há também o caso de funções que produzem resultados independentemente dos tipos de seus argumentos. Por exemplo, uma função de ordenação recebe como parâmetro um vetor de certos

valores e devolve o mesmo vetor com seus valores ordenados, não importando o tipo dos seus elementos. Exige-se apenas que eles sejam comparáveis entre si, conforme o critério de ordenação.

O conceito de classes genéricas oferece uma solução para implementação de código que funcionam bem com diversos tipos de dados e resolvem os problemas citados acima.

Exercícios

1. Mostre como objetos contêiner podem ser implementados em Java sem o uso de classes genéricas.
2. Compare a solução do exercício anterior com uma que use classes genéricas. Mostre os ganhos e as perdas.
3. Explique como métodos de classes genéricas podem ser sobrecarregados.

Notas bibliográficas

Deitel [10] apresenta uma introdução a classes genéricas de Java por meio de muitos e bons exemplos de programação.

Bertrand Meyer [40] discute vantagens e desvantagens do uso de genéricos e herança como mecanismos para favorecer reusabilidade.

Capítulo 14

Expressões-Lambda

Em Java, há apenas duas categorias de tipos para declaração de atributos de classe, variáveis de blocos e parâmetros de métodos: os tipos básicos e os tipos referência.

Em comandos de atribuição, um valor de tipo básico ou do tipo referência pode ser atribuído a uma variável. O mesmo ocorre na passagem de parâmetros a métodos, a qual é sempre por valor, i.e., o valor do argumento de chamada é copiado para o parâmetro formal correspondente. É como se fosse uma atribuição. Portanto, as regras de atribuição se aplicam à passagem de parâmetros.

E nomes de métodos não podem ser diretamente atribuídos a variáveis nem passados como parâmetros a outros métodos.

Apesar de esse cenário sugerir uma limitação na linguagem, passar uma funcionalidade como parâmetro a um método sempre foi possível em Java, embora isso requeira um certo malabarismo na programação.

A solução para contornar essa restrição é criar uma classe com o método que implementa a função que se deseja passar como parâmetro, alocar um objeto dessa classe e então passar a referência desse objeto como parâmetro ao método desejado. A partir da referência passada, pode-se ativar a função contida no objeto referenciado.

14.1 Interfaces funcionais

Para padronizar os procedimentos de passagem de funcionalidades a métodos via parâmetros, é conveniente implementá-los centrado na ideia de **interface funcional**, que é uma interface que tem apenas um método abstrato.

Uma interface funcional pode ter outros métodos, desde que sejam *defaults* ou estáticos. O ponto principal é que apenas um método da interface ainda dependa de implementação, e a intenção é prover um mecanismo alternativo e conciso para definir esse método.

Para ilustrar esse mecanismo, suponha que se deseja construir um programa, chamado `Selecionador`, que processe dados relativos a uma pessoa e realize uma seleção dos indivíduos analisados conforme os critérios de obesidade, maioridade e saúde aplicados objetos do tipo `Pessoa`, definido da seguinte forma:

```
public class Pessoa {
   public String nome;
   public float temp;
   public float peso;
   public int idade;
   public String profissão;
}
```

14.2 Sistema de triagem - versão I

Inicialmente, define-se a interface `Teste1`, que é usada como o primeiro exemplo de interface funcional.

Essa interface define o cabeçalho do método `teste` destinado a testar um ou mais atributos do parâmetro do tipo `Pessoa` que lhe for passado.

```
public interface Teste1 {boolean teste(Pessoa p);}
```

A partir da interface funcional **Teste1**, pode-se, conforme a demanda da aplicação, formar um conjunto de classes que a implementam, sendo que cada uma deve prover uma semântica própria para seu único método abstrato de nome **teste**.

Exemplos dessas classes são as apresentadas a seguir, as quais implementam o mesmo e único método abstrato de **Teste**, cada uma de uma forma distinta, considerando os critérios de seleção de pessoas anunciados para o problema proposto.

```
1 public classe TestaFebre1 implements Teste1 {
2    boolean teste(Pessoa p) {return (p.temp >= 37.0);}
3 }
4 public classe TestaObesidade1 implements Teste1 {
5    boolean teste(Pessoa p) {return (p.peso > 100.0);}
6 }
7 public classe TestaMaioridade1 implements Teste1 {
8    boolean teste(Pessoa p) {return (p.idade >= 18);}
9 }
```

A classe **Triagem1** abaixo faz uso da interface funcional **Teste1** para prover, entre outros, o serviço de seleção de pessoas sob os critérios definidos para o método **Teste1.teste**. Note que o método **selecione** recebe como parâmetro um objeto contendo o critério de seleção a ser aplicado a cada pessoa.

```
1 public class Triagem1 {
2   boolean selecione(Pessoa p, Teste1 c) {
3      return c.teste(p);
4   }
5   ...
6 }
```

E o programa principal **Seletor1** mostra como os métodos de nome **teste**, embutidos nos objetos da família de **Teste1**, são passados às diversas chamadas do método **selecione**.

```
1  public class Seletor1 {
2    public void static main(String... args) {
3       Triagem1 triagem = new Triagem1();
4       boolean b1, b2, b3;
5       Pessoa p = new Pessoa(); ...
6       Teste1 c1 = new TestaMaioridade1();
7       b1 = triagem.selecione(p, c1);
8       Teste1 c2 = new TestaObesidade1();
9       b2 = triagem.selecione(p, c2);
10      Teste1 c3 = new TestaFebre1();
11      b3 = triagem.selecione(p, c3); ...
12   }
13 }
```

O programa `Seletor1` cumpre seu papel, mas a interface funcional `Teste1` tem o inconveniente de o parâmetro de `teste` ser da classe `Pessoa`, e, portanto, de limitada reusabilidade.

14.3 Sistema de triagem - versão II

Para resolver o problema de reusabilidade apontado na versão I do sistema de triagem, a interface funcional é, agora, substituída por uma interface funcional genérica, como a seguinte declaração:

```
public interface Teste2<T> {boolean teste(T p);}
```

Como a interface `Teste2` não referencia uma classe específica, a definição de `Pessoa` pode ser feita somente quando necessária.

```
1  public class Pessoa {
2     public String nome, profissão;
3     public float temp, peso;
4     public int idade;
5  }
```

Essa alteração da interface `Teste1` implica na reimplementação de `TestaFebre1`, `TestaObesidade1`, `TestaMaioridade1` e `Triagem1`:

```
1  public classe TestaFebre2 implements Teste2<Pessoa> {
2     boolean teste(Pessoa p) {return (p.temp >= 37.0);}
3  }
4
5  public classe TestaObesidade2 implements Teste2<Pessoa> {
6     boolean teste(Pessoa p) {return (p.peso > 100.00);}
7  }
8
9  public classe TestaMaioridade2 implements Teste2<Pessoa> {
10    boolean teste(Pessoa p) {return (p.idade >=18);}
11 }
```

A classe `Triagem` deve ser devidamene ajustada para operar com o tipo parametrizado `Teste2<T>`:

```
1  public class Triagem2 {
2    boolean selecione(Pessoa p, Teste2<Pessoa> c) {
3       return c.teste(p);
4    }
5    ...
6  }
```

E o novo programa principal, `Seletor2`, mostra como os critérios de seleção, que estão embutidos nos objetos da família `Triagem2` são passados às chamadas do método `selecione`.

O artifício apresentado nesse programa, envolvendo interfaces funcionais e classes que a implementam, contorna a limitação inicial imposta pela linguagem, mas ainda tem o defeito de tornar o código verboso. Seria interessante evitar a criação de classes como `TestaMaioridade2`, `TestaObesidade2` e `TestaFebre2` usadas no programa abaixo.

```
1  public class Seletor2 {
2     public void static main(String... args) {
3     Triagem2 triagem = new Triagem2();
4     boolean b1, b2, b3;
5     Pessoa p = new Pessoa(); ...
6     Teste2<Pessoa> c1 = new TestaMaioridade2<Pessoa>();
7     b1 = triagem.selecione(p, c1);
8     Teste2<Pessoa> c2 = new TestaObesidade2<Pessoa>();
9     b2 = triagem.selecione(p, c2);
10    Teste2<Pessoa> c3 =  new TestaFebre2<Pessoa>()
11    b3 = triagem.selecione(p, c3);
12       ...
13    }
14 }
```

Uma solução com maior poder de expressão surgiu com Java 8, que introduziu o recurso de expressões-lambda, que destinam-se a reduzir um pouco a verbosidade observada acima pela eliminação da necessidade de se escrever as classes que implementam as interfaces funcionais necessárias e também os comandos de criação dos objetos dessas interfaces, os quais devem ser passados aos métodos.

Com expressões-lambda, o programador ainda deve definir a interface funcional necessária, mas as classes que implementam essa interface não precisam ser diretamente programada nem os objetos que lhes correspondem ser criados explicitamente. Tudo isso é feito pelo compilador!

14.4 Sintaxe das expressões-lambda

Uma expressão-lambda é uma função anônima e tem o seguinte formato:

```
(<parâmetros>) -> <corpo>
```

onde `<parâmetros>` é uma lista possivelmente vazia de declarações de parâmetros, separadas por vírgulas, e `<corpo>` é uma expressão ou um bloco de comandos.

Por exemplo, cada uma das linhas abaixo é uma expressão-lambda:

```
1  () -> null;
2  (int x, int y) -> x + y
3  (float x, float y) -> x * y
4  (int a, long b) -> (long) a + b
5  (x, y) -> x/y
6  (z) -> z + 1
7   z -> z + 1
8  (String s) -> {s = s + " "; system.out.print(s);}
9  (String s) -> system.out.print(s)
10 (s) -> system.out.print(s);
11 (Pessoa p) -> {return p.nome;}
12 (Thread t) -> t.start();
```

Os parênteses que envolvem os parâmetros formais podem ser omitidos no caso de se ter apenas um parâmetro, e os seus tipos também podem ser omitidos, se puderem ser inferidos no momento do uso da expressão-lambda, que deve ser vinculado a uma interface funcional.

O corpo de uma expressão-lambda que contém apenas uma chamada a um método dispensa o uso das chaves { }.

14.4.1 Uso de expressões-lambda

Suponha as seguintes definições de interfaces funcionais:

```
interface T1 {int op(int x, int y);}
interface T2<E> {E op(E x, E y);}
```

Uma expressão-lambda pode ser atribuída à variável cujo tipo denota uma interface funcional cuja operação tem tipo compatível.

Essa operação de atribuição embute uma série de providências, a ser realizada internamente, que o trecho de código a seguir pretende esclarecer:

```
1  public class Atribuição {
2     public static void main(String [ ] args) {
3        T1 a = (int m, int n) -> m * n;
4        T2<Integer> b = (Integer x, Integer y) -> x + y;
5        T2<Integer> c = (r, s) -> r - s;
6        int z1 = a.op(2,5);  // z1 = 10
7        int z2 = b.op(1,2);  // z2 = 3
8        int z3 = c.op(10,5); // z3 = 5
9        System.out.println(z1 + z2 + z3);  // saida: 18
10    }
11 }
```

O comando da linha 3 acima tem o efeito de criar uma classe anônima que implementa a interface `T1` definindo que o corpo do método abstrato `op` tem a mesma semântica do código dado pela expressão-lambda indicada no comando, e, a seguir, alocar um objeto dessa classe anônima e atribuir seu endereço à referência `a`.

As linhas 4 e 5, que têm efeito similar a da 3, atribuem os endereços dos objetos criados às referências `b` e `c`, respectivamente.

14.5 Sistema de triagem - versão III

A aplicação proposta no início deste capítulo para implementar um sistema de triagem, agora refeita sob a denominação `Seletor3` é apresentada a seguir usando os recursos de expressões-lambda.

Para facilitar a leitura, e mostrar todo o código da presente solução e facilitar a comparação com as outras duas apresentadas para o mesmo problema, repetem-se aqui as definições da interface funcional genérica `Teste2` e da classe `Triagem2`, convenientemente renomeadas para `Teste3` e `Triagem3`, respectivamente.

```
1  interface Teste3<T> {boolean teste(T p);}
2  class Pessoa {
3     public String nome;
4     public float temp;
5     public float peso;
6     public int idade;
7     public String profissao;
8  }
9  class Triagem3 {
10   boolean selecione(Pessoa p, Teste3<Pessoa> c) {
11      if (c.teste(p)) return true;
12      else return false;
13   }
14 }
15 public class Seletor3 {
16   public static void main(String ... args) {
17      Triagem3 triagem = new Triagem3();
18      boolean b1, b2, b3;
19      Pessoa p = new Pessoa(); ...
20      Teste3<Pessoa> c1 = p1 -> {return (p1.idade >= 18);};
21      b1 = triagem.selecione(p, c1);
22      Teste3<Pessoa> c2 = p2 -> {return (p2.peso > 100.0);};
23      b2 = triagem.selecione(p, c2);
24      Teste3<Pessoa> c3 = p3 -> {return (p3.temp >= 37.0);};
25      b3 = triagem.selecione(p, c3);
26      ...
27   }
28 }
```

Com certeza, o código desta versão do programa de triagem é mais compacto devido ao uso de expressões-lambda.

14.6 Acesso ao escopo envolvente

Expressões-lambda não introduzem um novo nível de escopo de variáveis. Em seus corpos, além dos parâmetros, pode-se ter acesso

às variáveis finais, assim declaradas, ou efetivas, de métodos e classes envolventes, e as declarações que ocorrem em uma expressão-lambda são tratadas como se tivessem sido declaradas no escopo imediatamente envolvente.

O seguinte exemplo mostra como variáveis do escopo envolvente de uma expressão-lambda são acessadas.

```
1  interface I {void f(int y);}
2
3  class A {
4    public int r1, r2, r3, r4 = 20;
5    int x = 3;
6    public class B {
7      public final int x = 2;
8        void g(int x) {
9           r4 = 10;
10          I c = (y) -> {
11              r1 = x; r2 = y; r3 = this.x; r4 = A.this.x;
12          };
13          c.f(x+1);
14       }
15   }
16 }
17
18 public class Lambda {
19   public static void main(String... args) {
20     A a = new A();
21     A.B h = a.new B();
22     h.g(1);
23     System.out.print  (" r1 = " + a.r1 + " r2 = " + a.r2);
24     System.out.println(" r3 = " + a.r3 + " r4 = " + a.r4);
25   }
26 }
```

O programa `Lambda` acima imprime

```
r1 = 1 r2 = 2 r3 = 3 r4 = 4
```

14.6.1 Variáveis em expressões-lambda

Campos, variáveis locais a bloco e parâmetros que são declarados `final` são ditas variáveis finais, podendo ser iniciadas apenas uma vez no fluxo de execução. As finais efetivas são as que, embora não sejam assim declaradas, têm o mesmo comportamento que o de uma declarada final. As variáveis usadas no corpo de uma expressão-lambda devem ser finais ou finais efetivas, como ilustra o programa:

```
1  interface I {int h();}
2  class A {
3     void f(I g) {System.out.print(g.h());}
4     public void g1(final int x) {
5        int y = 1;
6        f(() -> x+y);  // y é final efetiva
7     }
8     public void g2(int x) {
9        final int y;
10       y = 1;
11       f(() -> x+y);  // x é final efetiva
12    }
13    public void g3(int x) {
14     int y;
15     if (x==0) y = 1; else y = 2;
16     f(() -> x+y); // y é final efetiva
17    }
18     public void g4(int x) {
19        int y;
20        if (x==0) y = 1; else y = 2;
21        y = 3;          // y deixa de ser final efetiva
22        f(() -> x+y);  // Erro por usar y
23    }
24 }
```

As expressões-lambda da linhas 6, 11 e 16 estão corretas, pois os

`x` e `y` nelas referenciados são variáveis finais efetivas. A expressão da linha 22 está errada, porque `y` não é final efetiva.

14.7 Compatibilidade de tipos

Uma expressão-lambda é compatível com um tipo `T` se `T` for uma interface funcional e a expressão-lambda for *congruente* com o tipo funcional do método abstrato de `T`.

Ser congruente significa ter parâmetros compatíveis em número e tipos, e tipos de retorno também compatíveis.

Uma expressão-lambda λ que seja compatível com uma interface funcional `T` que especifica o método abstrato `f` pode ser usada como:

- lado direto de atribuição a uma variável do tipo `T`, como em `T t =` λ;
- argumento de chamada correspondente a parâmetro formal do tipo `T` de um método, e.g., `z.m(`λ`)`, para `void m(T x){...}`;
- operando de um *casting* para `T`, como em `Object t = (T)` λ.

14.8 Conclusão

Expressões-lambda são uma notação que simplifica o processo de declarar uma classe que implementa uma interface funcional, criar um objeto dessa classe e atribuir sua referência a alguma variável do programa ou a parâmetro de alguma função. Tudo que tem que ser feito é atribuir a expressão-lambda que implementa o corpo do método abstrato da interface ao objeto desejado, seja ela uma variável ou um parâmetro.

Exercícios

1. Qual seria o ganho de poder de expressão de Java obtido pelo mecanismo de expressões lambda?

Notas bibliográficas

O conceito de expressões-lambda é bem discutido nos tutoriais da página da Oracle (https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)

Capítulo 15

Coleções

Uma coleção é um contêiner de objetos de um dado tipo. As interfaces `Collection` e `Map` são implementadas por classes que manipulam coleções de objetos de mesmo tipo, provendo pelo menos as operações de armazenar, recuperar e manipular esses objetos. O conjunto de interfaces e classes que tratam de coleções é chamado de Arcabouço de Coleções (*Collections Framework*).

As interfaces desse arcabouço estão organizadas em categorias afins, que formam uma hierarquia de interfaces, cada uma com definições próprias dos contratos que disciplinam o uso e execução das operações herdadas, como mostra a Fig. 15.1.

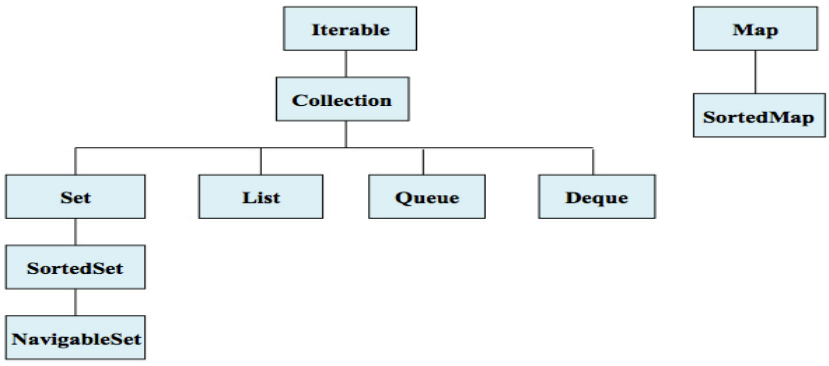


Figura 15.1 Hierarquias do Arcabouço das Coleções

Uma coleção nesse arcabouço é um tipo abstrato de dados definido pela interface `Collection` ou por `Map`, a partir das quais são implementadas diversas classes, como `Stack` e `LinkedList`,

modelando importantes tipos de estruturas de dados. Essas classes implementam interfaces que definem diversas operações, como `add`, `equals` e `hashcode`, que têm significado especial em cada implementação.

As diversas classes do arcabouço *Collections* formam a hierarquia de interfaces da Fig. 15.1, que lá estão para impor restrições nos contratos das operações herdadas.

Os tipos implementados no arcabouço são todos tipos abstratos de dados, e, por isso, permitem seu uso sem direito de acesso direto às suas representações, reduzindo o esforço de programação e garantindo alto grau de reúso.

15.1 Iteradores

No topo da hierarquia de `Collection` está a interface

```
public interface Iterable {Iterator<E> iterator();}
```

que associa a coleções o conceito de objetos iteradores, os quais, como o nome indica, servem para percorrer e recuperar, eficientemente, um a um, os objetos contidos em uma dada coleção. Para isso, toda classe da hierarquia de `Collection` implementa uma operação denominada `iterator()` para criar um objeto iterador que permita percorrer os elementos da coleção.

Iteradores são objetos do tipo `Iterator`, o qual é definido pela seguinte interface:

```
1 interface Iterator<E> {
2     boolean hasNext();
3     E next() throws IllegalStateException;
4     void remove();
5 }
```

A operação `hasNext()` de um objeto do tipo `Iterator` informa se há ou não elementos a ser recuperados da coleção a que o iterador

está associado. A operação `next()` retorna o próximo elemento da lista de objetos da coleção. Caso o próximo elemento não exista, a exceção `IllegalStateException` é lançada. E terceira operação de `Iterator`, o método `remove()`, apaga da coleção associada o último elemento retornado.

O núcleo das coleções de Java, conforme mostra a Fig. 15.1, inclui duas interfaces principais: `Collection` e `Map`.

A interface `Collection` é a raiz dos tipos `Set`, `List`, `Queue` e `Deque`, que lhe são extensões diretas, e associadas à interface `Map`, há também nessa biblioteca as classes `SortedMap`, `EnumMap`, `HashMap`.

15.2 Interface `Collection`

A interface genérica `Collection<E>`, onde `<E>` é o tipo genérico dos objetos administrados pela coleção, é definida pelas seguintes operações:

```
1  interface Collection<E> extends Iterable<E> {
2     int size();
3     boolean isEmpty();
4     boolean contains(Object elem);
5     boolean containAll(Collection<?> col);
6     Object[] toArray();
7     <T> T[] toArray(T[] elem) throws ArrayStoreException;
8     boolean add(E elem);
9     boolean addAll(Collection<? extends E> col);
10    boolean remove(Object elem);
11    boolean removeAll(Collection<?> col);
12    boolean retainAll(Collection<?> col);
13    void clear();
14 }
```

As operações aplicáveis a um objeto `this` do tipo `Collection` têm o seguinte significado, que pode sofrer algumas restrições dependendo da sua classe específica:

- `int size()`:
  informa o tamanho da coleção `this`.
- `boolean isEmpty()`:
  informa se a coleção `this` está ou não vazia.
- `boolean contains(Object elem)`:
  Se `elem == null`, informa se há um objeto `null` na coleção `this`, senão informa se existe na coleção `this` um objeto `obj`, tal que `elem.equals(obj)`.
- `boolean containAll(Collection<?> col)`:
  retorna verdadeiro se os objetos em `col` estiverem na coleção `this`.
- `Iterator<E> iterator()`:
  retorna um objeto que percorre os elementos da coleção `this` segundo a interface `iterador` definida acima. Operação herdada de `Iterable`.
- `Object[] toArray()`:
  retorna um arranjo com todos os objetos presentes na coleção `this`.
- `<T> T[] toArray(T[] elementos)`:
  retorna um arranjo com todos os elementos da coleção `this`. Se o arranjo `elementos` não tiver tamanho suficiente, seu tamanho será convenientemente expandido. Se for maior que o tamanho da coleção, a parte final será preenchida com `null`. A exceção `ArrayStoreException` é lançada se o tipo dos elementos da coleção `this` não for compatível com o tipo de `elementos`.
- `boolean add(E elem)`:
  adiciona `elem` na coleção `this`, retornando `false` quando a

coleção não admitir inserção de objeto repetido. Restrições como essa são impostas por subinterfaces de `Collection`.

- `boolean addAll(Collection<? extends E> col)`:
  adiciona todos objetos da coleção `col` na coleção `this`.
- `boolean remove(Object elem)`:
  remove uma ocorrência do objeto `elem` da coleção `this`, inclusive no caso de o objeto ser `null`, retornando `false` apenas quando `elem` não estiver presente.
- `boolean removeAll(Collection<?> col)`:
  remove todos os objetos da coleção `col` da coleção `this`.
- `boolean retainAll(Collection<?> col)`:
  remove da coleção `this` os objetos que não estão na coleção dada `col`.
- `void clear()`:
  limpa coleção `this`, removendo todos os elementos.

15.3 Interface `Set`

A interface `Set<E>` representa um conjunto de objetos do tipo `E`, no qual não há elementos repetidos nem ordenados, sendo permitido o elemento `null`. As operações de classes que implementam `Set` diretamente devem fazer valer essas restrições.

A igualdade de dois elementos `a` e `b` de um conjunto é definido pela operação `a.equals(b)`.

A interface `Set` é estendida pela interface `SortedSet`, com o propósito de formalizar a opção de se criar classes que impõem uma ordenação sobre os elementos do conjunto. Classes que implementam diretamente a interface `Set` são `HashSet`, `LinkedHashSet` e `TreeSet`, presentes no arcabouço. Objetos declarados como `Set` de objetos do tipo `E` podem ser criados por uma das seguintes

formas:

```
Set<E> s1 = new HashSet<E>();
Set<E> s2 = new LinkedHashSet<E>();
Set<E> s3 = new TreeSet<E>();
```

15.3.1 Classe `HashSet`

A classe **`HashSet<E>`** define conjuntos de elementos de tipo **`E`** que são armazenados por meio de uma tabela *hash*, e a ordem de armazenamento dos seus elementos não tem padrão definido.

Os elementos de um objeto **`HashSet`** são percorridos em uma ordem fora do controle do usuário, pois não há regras de ordenação predefinida. O seguinte exemplo ilustra o uso classe **`HshSet`**:

```
1  import java.util.*;
2  public class TestaHashSet1 {
3     public static void main(String args[]) {
4        HashSet<String> c = new HashSet<String>();
5        c.add("Felipe");
6        c.add("Mariza");
7
8        Iterator<String> it1 = c.iterator();
9        while(it1.hasNext())
10          System.out.print(" it1: " + it1.next());
11       c.add("Carolina");
12       c.add("Carolina");
13       c.add("Patricia");
14
15       Iterator<String> it2 = c.iterator();
16       System.out.println();
17       while(it2.hasNext())
18          System.out.print(" it2: " + it2.next());
19       System.out.println();
20    }
21 }
```

O programa `TestaHashSet1` acima, que define dois iteradores: `it1` na linha 8 e `it2` na linha 15, imprime o seguinte texto:

```
it1: Mariza it1: Felipe
it2: Mariza it2: Felipe it2: Patricia it2: Carolina
```

Observe que:

- a segunda inclusão de `"Carolina"` na linha 12 foi convenientemente ignorada;
- a ordem de impressão não tem relação com a ordem de inclusão de elementos no conjunto;
- entre a captura do iterador do conjunto e os usos do `next()` associado, o conteúdo desse conjunto não pode ser alterado. Se isso não for observado, a execução é cancelada. Por exemplo, o programa `TestaHahSet2` abaixo comprova essa restrição de uso do iterador.

```
1  import java.util.*;
2  public class TestaHashSet2 {
3     public static void main(String args[]) {
4        HashSet<String> c = new HashSet<String>();
5        c.add("Felipe");
6        c.add("Mariza");
7        Iterator<String> it = c.iterator();
8        c.add("Carolina");
9        c.add("Patricia");
10       while(it.hasNext())
11          System.out.print(" it: " + it.next());
12       System.out.println();
13    }
14 }
```

O programa acima compila sem erro, mas ao se tentar executá-lo obtém-se a mensagem relativa ao uso do `it.next()` na linha 11:

```
Exception in thread "main"
          java.util.ConcurrentModificationException
at java.base/java.util.HashMap$HashIterator.
                        nextNode(HashMap.java:1494)
at java.base/java.util.HashMap$KeyIterator.
                        next(HashMap.java:1517)
at TestaHashSet2.main(TestaHashSet2.java:11)
```

15.3.2 Classe `LinkedHashSet`

A classe `LinkedHashSet<E>` estende a classe `HashSet<E>` e implementa a interface `Set<E>`, usando uma estrutura interna baseada em lista duplamente encadeada. Por usar uma lista, os elementos possuem uma ordem interna, que é usada pelo seu iterador.

15.3.3 Interface `SortedSet`

`SortedSet<E>` é uma especialização da interface `Set<E>` com o objetivo de impor uma ordenação ascendente aos elementos do conjunto subjacente. Todos os elementos do conjunto, e.g., `e1` e `e2`, devem implementar a interface `Comparator`, de forma a ser comparáveis pela operação `e1.compareTo(e2)`. Os iteradores de `HashSet` percorrem os elementos do conjunto em ordem crescente.

15.3.4 Classe `TreeSet`

A classe `TreeSet<E>` implementa a interface `NavigableSet<E>`, que é uma subinterface de `Set<E>`, e caracteriza-se por usar uma árvore para armazenar os elementos do conjunto subjacente. O uso de árvore torna o tempo de recuperação de elementos do conjunto em ordem ascendente eficiente.

15.4 Interface List

A interface List<E> estende Collection<E> diretamente e estabelece que classes que implementem esse tipo de coleção devem armazenar seus elementos em lista de objetos ordenados por sua posições. As condições contratuais impostas às operações sobre os elementos armazenados na coleção permitem que se possa ter controle da posição de inserção dos elementos na lista que implementa a coleção.

As classes da família List são ArrayList, LinkedList, Stack e Vector.

15.4.1 Classe ArrayList

A classe ArrayList<E> implementa List<E> por meio de um arranjo. Essa representação tem efeito direto no desempenho de suas operações, pois provê acesso eficiente a seus elementos via indexação de arranjo, executa com eficiência inserções e remoções de elementos no fim da lista, mas apresenta um custo mais elevado nas operações de inserção ou remoção no meio lista, devido a necessidade de movimentação de elementos.

15.4.2 Classe LinkedList

A classe LinkedList<E> implementa List<E> por meio de uma lista duplamente encadeada. Essa representação tem efeito direto no desempenho de suas operações, pois provê operações eficientes para inserções e remoções de elementos tanto no início como no fim da lista, mas apresenta um custo mais elevado nas operações de inserção ou remoção no meio lista, devido à necessidade de se percorrer a lista para localizar a correta posição de inserção ou remoção.

15.4.3 Classe `Vector`

A classe `Vector<E>` implementa a interface `List<E>` por meio de um arranjo de tamanho flexível, que tem seu tamanho alterado automaticamente conforme a demanda, nos termos especificados no momento de sua criação. O uso de `Vector` mostra-se muito útil quando não há como prever o tamanho de arranjo com o qual trabalha-se.

As principais operações de `Vector`, dentre muitas outras, são:

- `public Vector()`:
  aloca um espaço para 10 elementos.
- `public Vector(int capacidade)`:
  aloca espaço para até `capacidade` elementos.
- `public Vector(int capacidadeInicial, int incremento)`:
  aloca espaço inicial para até `capacidadeInicial` elementos e define que, se vier a ser necessário, esse espaço deve ser expandido automaticamente conforme o valor de `incremento`.
- `public int capacity()`:
  informa capacidade corrente do vetor.
- `public void ensureCapacity(int minimo)`:
  aumenta a capacidade do vetor para poder conter pelo menos `minimo` elementos.
- `public Iterator<E> iterator()`:
  retorna um iterator para percorrer a lista de elementos do vetor.
- `public final void addElement(E elem)`:
  adiciona `elem` no fim do vetor.
- `public final void add(int indice, E elem)`:
  insere `elem` na posição especificada por `indice`.
- `public final void setElementAt(E elem, int indice)`:

faz o elemento na posição `indice` do vetor ter o valor de `elem`.

- `public final void removeElementAt(int indice)`:
  remove do vetor o elemento da posição `indice`.
- `public final E ElementAt(int indice)`:
  retorna o elemento da posição `indice`.

O programa `Vector1` abaixo cria um vetor de nomes de 10 posições, insere nesse vetor quatro nomes e imprime seu conteúdo via um iterador. Confira:

```
1  import java.util.*;
2  public class Vector1 {
3     public static void main(String args[]) {
4        Vector<String> v = new Vector<String>(10);
5        v.add("Felipe");
6        v.add("Mariza");
7        v.add("Carolina");
8        v.add("Patricia");
9        Iterator<String> it = v.iterator();
10       while(it.hasNext())
11          System.out.print(" " + it.next());
12       System.out.println();
13    }
14 }
```

imprime `Felipe Mariza Carolina Patricia`

15.4.4 Classe `Stack`

A classe `Stack` é uma subclasse de `Vector` que implementa uma estrutura de pilha, portanto tem todas as operações de `Vector`. Entretanto, como a ideia é operar a estrutura como uma pilha e não como um vetor, `class Stack<E>` oferece as seguintes operações:

- `public Stack()`:
  aloca uma pilha vazia.

- `public E push(E item)`:
  empilha `item`.
- `public E pop()`:
  desempilha item e o retorna.
- `public E peek()`:
  retorna o item no topo da pilha.
- `public boolean empty()`:
  informa se a pilha está ou não vazia.
- `public int search(Object o)`:
  retorna a posição do objeto `o` na pilha.

As demais operações de `Stack<E>` herdadas de `Vector<E>` devem ser usadas criteriosamente para respeitar a individualidade das pilhas, pois são as operações de um tipo que o definem.

15.5 Interface `Queue`

A interface `Queue<E>` é uma extensão de `Collection<E>` que define as operações típicas de processamento de filas FIFO (*first-in-first-out*) ou de filas de prioridades. Nas filas FIFO, o elemento de maior prioridade é o presente na fila há mais tempo, enquanto que nas filas de prioridade é o elemento definido segundo algum critério de comparação de valores armazenados nos elementos da fila. As principais operações previstas para processamento de filas são:

- `boolean add(E item)`:
  insere o elemento `item` na fila e retorna `true` se operação teve sucesso. Se não, levanta a exceção `IllegalStateException`.
- `boolean offer(E item)`:
  insere o elemento `item` na fila se isso for possível. Retorna `true` se teve sucesso, e `false`, caso contrário.

- `public E remove()`:
  remove e retorna o elemento de maior prioridade da fila, chamado de cabeça da fila.
- `public E peek()`:
  retorna a cabeça da fila, sem removê-la.
- `public E poll()`:
  remove e retorna a cabeça da fila ou retorna `null` se a fila estiver vazia.
- `public boolean empty()`:
  informa se a fila estão ou não vazia. Operação herdada de `Collection`.

As demais operações de `Queue<E>` herdadas de `Collection<E>` devem ser usadas criteriosamente para respeitar a individualidade das filas, pois são as operações de um tipo que o definem.

Algumas classes do arcabouço das coleções de Java que implementam a interface `Queue` são `AbstractQueue`, `PriorityQueue` e `ArrayBlockingQueue`.

15.6 Interface `Deque`

A interface `Deque` é uma extensão de `Queue`, a qual acrescentam-se operações para inserir e remover elementos nos dois extremos da fila. O nome `deque` é um acrônimo de *double ended queue*.

As principais e típicas operações de classes que implementam `Deque<E>` no arcabouço `Collection` são:

- `void addFirst(E item)`:
  adicional `item` na frente da fila, respeitada sua capacidade.
- `E removeFirst()`:
  remove o elemento no início da fila e o retorna.

- `void addLast(E item)`:
  adiciona `item` no fim da fila, respeitada sua capacidade.
- `removeLast(E item)`:
  remove e retorna o último iem da fila.
- `boolean isEmpty()`:
  informa se a fila está ou não vazia.
- `int size()`:
  retorna o número de itens na fila.

Classes do arcabouço das coleções de Java que implementam a interface `Deque` são `ArrayDeque`, `ALinkedBlockingDeque` e `ConcurrentLinkedDeque`.

Um exemplo a seguir mostra o uso da classe `ArrayDeque` para construir uma fila *deque* de nomes:

```
1  import java.util.*;
2  public class TestaDeque2 {
3     public static void main(String... args) {
4        Deque<String> d = new ArrayDeque<String>();
5        d.addFirst("Carolina");   // 3
6        d.addFirst("Patricia");   // 2
7        d.addLast("Felipe");      // 4
8        d.addFirst("Mariza");     // 1
9        Iterator<String> it = d.iterator();
10       while (it.hasNext()) System.out.println(it.next());
11    }
12 }
```

O resultado impresso por `TestaDeque2` é:

```
Mariza
Patricia
Carolina
Felipe
```

15.7 Interface Map

A interface Map<K,V> define as operações de uma estrutura que relaciona chaves do tipo K a valores de tipo V.

A principal classe que implementa Map<K,V) é HashMap, que provê um mapeamento de chaves a valores via algoritmos de *hashing* de objetos do tipo K, os quais devem implementar os métodos hashcode e equals.

A estrutura implementada por essa classe é muito flexível e pode ser manipulada por cerca de 30 operações.

15.8 Conclusão

Este capítulo apresentou ao leitor uma introdução ao rico arcabouço de coleções de Java, descrevendo um conjunto bem limitado de interfaces e classes, apenas o suficiente para a escrita dos primeiros programas.

O uso dos tipos definidos nesse arcabouço é altamente recomendado para reduzir o esforço de implementação e aumentar a qualidade do código produzido pelo uso de componentes de software já testados e que garantidamente funcionam.

Exercícios

1. Qual é a relação entre classes e tipos abstratos de dados?

2. O que define um tipo abstrato de dados?

3. Os projetistas de Java decidiram criar a hierarquia Collections da Fig. 15.1, a qual força um grande compartilhamento de operações entre os todos os tipos vinculados a essa hieraquia. Avalie o impacto dessa decisão considerando que são as operações que deferenciam os tipos.

Notas bibliográficas

Para o entendimento mais aprofundado do tema a melhor referência é a pagina da Oracle [43], onde há ricos tutoriais e definições claras de todas as operações definidas para as classes desse arcabouço.

Coleções é um conceito bem discutido em praticamente todos os textos sobre a linguagem Java, e em particular nos livros de K. Arnold, J. Gosling e D. Holmes [2, 3] e dos Deitels [10].

Capítulo 16

Reflexão Computacional

Há situações em que o acesso, durante a execução de um programa, a informações sobre a estrutura interna de seus dados ou do seu código se faz necessário ou conveniente. A atividade de se focalizar nos elementos intrínsecos de um programa, solicitar-lhe informações de si e manipulá-las é chamada de **reflexão computacional**.

A toda classe, durante a execução, há um objeto associado do tipo `Class` contendo sua descrição, que compreende, entre outros dados, seu nome, nomes de seus campos, métodos e construtoras, os modificadores de acesso de cada membro e a identificação de superclasses e interfaces.

Pratica-se reflexão computacional, quando se faz acesso durante a execução a essas informações sobre um dado objeto e sua classe. Para isso, deve-se recuperar o objeto do tipo `Class` associado ao objeto ou a classe que se desejar inspecionar.

Java oferece diversos recursos para se obter o objeto do tipo `Class` desejado, como o método `getClass` da classe `Object`, o atributo estático `class` associado a todo tipo e os métodos `forName`, `getInterfaces`, `getSuperClass`, `getClasses`, `getEnclosingClass`, `getDeclaringClass`, `getDeclaredClass` e `getComponentType`, todos da classe `Class`.

16.1 Operação `getClass`

Considere o problema de verificar a compatibilidade dois objetos cujos tipos pertencem a uma mesma hierarquia que tem como raiz uma classe `A` que é estendida por uma classe `B`. Dois objetos `x` e `y` são compatíveis se ambos são de mesmo tipo dinâmico e os campos privados `A.i` e `B.j` de `x` forem iguais aos correspondentes de `y`. No caso de o tipo dinâmico ser `A`, basta a igualdade dos campos `A.i`.

O caráter polimórfico das referências e as restrições de visibilidade a membros de classe exigem um certo cuidado na implementação do método, e.g., `compatível`, que determina a compatibilidade do objeto corrente, `this`, e o objeto a ele passado como parâmetro, conforme ilustram as classes `A` e `B`, apresentadas a seguir.

```
1  public class A {
2    private int i;
3    public A(int i) {this.i = i;}
4    public boolean compatível(A a) {
5      if (a!=null && this.getClass()==a.getClass()){
6              return (this.i == a.i);
7        } else  return false;
8     }
9  }
10 public class B extends A {
11   private int j;
12   public B(int i, int j) {super(i); this.j = j;}
13   public boolean compatível(A a) {
14     if (a!=null && this.getClass()==a.getClass()) {
15           B b = (B) a;
16           return (super.compatível(a) && (this.j==b.j));
17       } else return false;
18    }
19 }
```

Durante a execução do método **compatível** declarado na linha 4, o objeto apontado pela referência **this** é garantidamente do tipo **A** ou do de um de seus descendentes. Assim, o parâmetro **a** de **compatível** tem que ter tipo dinâmico igual ao tipo de **this** para que possa haver compatibilidade entre os objetos apontados por **this** e **a**.

O método **getClass** da classe **Object**, usado duas vezes em cada uma das linhas 5 e 14, retorna o endereço do objeto descritor da classe do objeto receptor.

Como somente há um objeto descritor alocado para cada classe encontrada no programa, e classes distintas têm objetos descritores distintos, o teste **(this.getClass() == a.getClass())** permite determinar se **this** e **a** apontam ou não para o mesmo descritor de classe.

A validade da conversão de tipo da linha 15 é garantida pelo teste feito na linha anterior, que assegura que o objeto apontado pela referência **a** é realmente do tipo **B**.

Note que dentro da classe **B** somente pode-se comparar os campos nela visíveis, no caso, somente os valores do campo **j** dos dois objetos. Para comparar os elementos da parte **A** de objetos do tipo **B**, deve-se ativar, via **super**, a versão do método **compatível** definido na superclasse de **B**, na forma indicada na linha 16.

16.2 Atributo `class` dos tipos

Uma segunda forma de obter o endereço do descritor **Class** de uma classe é via o atributo estático **class**, que é associado a todo tipo, inclusive os primitivos.

Por exemplo, o comando

```
Class c = String.class
```

captura o descritor da classe **String**.

Para obter o objeto `Class` de tipos primitivos e de arranjos, procede-se como em:

```
Class c1 = int.class;
Class c2 = Integer.class;
Class c3 = int[][].class;
```

O tipo do objeto `Class` associado a tipos primitivos é o mesmo correspondente ao de sua classe invólucro. No exemplo acima, `c1` e `c2` denotam o mesmo objeto `Class`.

16.3 Operações de `Class`

A classe `java.lang.Class<T>` é uma classe genérica, declarada conforme o seguinte esquema:

```
public final class Class<T> extends Object
   implements Serializable, GenericDeclaration,
                          Type, AnnotatedElement {
   ...
   public static Class<?> forName(String nome) { ...}
   ...
   public String toString() { ... }
}
```

Cada objeto `Class` é então uma instância do tipo parametrizado conforme a classe que ele representa. Por exemplo, o tipo associado à `String.class` é `Class<String>`.

A seguir, um pequeno subconjunto das operações públicas de `Class<T>` estão anunciadas e brevemente descritas. Recomenda-se uma visita à página da Oracle [43] para ver a lista completa da operações dessa classe.

- `String getName()`:
  retorna o nome da classe do objeto `this`.
- `Class<? super T> getSuperClass()`:
  retorna objeto `Class` da superclasse de `this`.

- `Class<?>[] getInterfaces()`:
  retorna um arranjo de objetos `Class` que informa as interfaces implementadas por `this`.
- `boolean isInterface()`:
  informa se o objeto `Class` é uma interface.
- `boolean isPrimitive()`:
  informa se o objeto `Class` é um tipo primitivo.
- `boolean isArray()`:
  informa se o objeto `Class` é uma classe arranjo.
- `static Class forName(String className)`:
  retorna o objeto que representa a classe de nome `className`.
- `T newInstance()`:
  retorna uma nova instância da classe do objeto `this`.
- `Field[] getFields()`:
  retorna um arranjo contendo os objetos `Field` dos campos públicos.
- `Field[] getDeclaredFields`:
  retorna um arranjo contendo os objetos `Field` de todos os campos.
- `Method[] getMethods()`:
  retorna um arranjo contendo os objetos `Method` dos métodos públicos.
- `Method[] getDeclaredMethods`:
  retorna um arranjo contendo os objetos `Methods` de todos os métodos.
- `Constructor<?>[] getConstructors()`:
  retorna um arranjo contendo os objetos `Construtor` que fornece os construtores públicos.
- `Package getPackage()`:
  retorna o pacote desta classe.

- `String toString()`:
  retorna o resultado da conversão do objeto **`this`** em uma cadeia de caracteres.

Alguns exemplos de uso das operações de reflexão:

```
1  ...
2  Empregado e;
3  Class c = e.getClass();
4  Methods [] m = c.getMethods();
5  ...
6  String nome = "Empregado";
7  Class c = Class.forName(nome);
8  Methods [] m = c.getMethods();
9  Object x = c.newInstance();
10 ...
11 Empregado e = new Empregado("José da Silva",data);
12 Class c = e.getclass();
13 System.out.println(c.getName());
14 ...
```

as quais têm a seguinte semântica:

- Linhas 2 a 4:
  declara um objeto do tipo **`Empregado`**, obtém seu descritor de classe e com ele obtém seus métodos.
- Linhas 6 a 9:
  a partir da cadeia de caracteres que denota o nome de uma classe, obtém o descritor da classe com esse nome, em seguida, recupera seus métodos e depois aloca um objeto da classe obtida.
- Linhas 11 a 13:
  cria um objeto da classe **`Empregado`**, obtém seu descritor de classe e depois imprime **`Empregado`**.

16.4 Inspeção de classes

O pacote `java.lang.reflect` incorpora um conjunto de classes que descrevem as informações sobre a constituição de objeto da classe `Class`. As principais classes desse pacote são `Field`, `Method`, `Modifier` e `Constructor`. Objetos dessas classes podem ser recuperados pelos seguintes métodos da classe `Class`:

- `Field[] getFields()`
- `Field[] getDeclaredFields`
- `Method[] getMethods()`
- `Method[] getDeclaredMethods`
- `Constructor<?>[] getConstructors()`

Operações definidas pelas classes `Field`, `Method` e `Constructor`, aplicáveis a objetos dessas três classes incluem as seguintes:

- `Class<?> getDeclaringClass()`:
  retorna o objeto `Class` da classe que define o construtor, método ou campo corrente.
- `int getModifiers()`:
  retorna um inteiro que descreve os modificadores desse construtor, método ou campo. A classe `Modifier` provê operações para analisar o inteiro retornado.
- `String getName()`:
  retorna `String` que é o nome do construtor, método ou campo.
- `Class<?> getType`:
  retorna um objeto `Class` que representa o tipo de um campo
- `Class<?> getReturnType`:
  retorna um objeto `Class` que representa o tipo do valor de retorno de método.
- `Class<?>[] getParameterTypes`:
  retorna um arranjo de objetos `Class` que representam os tipos dos parâmetros desse construtor ou método.

- `Class<?>[] getExceptionTypes()`:
  retorna um arranjo de objetos `class` que representam os tipos de exceções levantadas pelo método.

As operações públicas da classe `Modifier`, cujos nomes são autoexplicativos, e que destinam-se a interpretar o inteiro retornado pela operação `getModifiers` são os seguintes:

- `static java.lang.String toString(int);`
- `static boolean isInterface(int);`
- `static boolean isPrivate(int);`
- `static boolean isTransient(int);`
- `static boolean isStatic(int);`
- `static boolean isPublic(int);`
- `static boolean isProtected(int);`
- `static boolean isFinal(int);`
- `static boolean isSynchronized(int);`
- `static boolean isVolatile(int);`
- `static boolean isNative(int);`
- `static boolean isAbstract(int);`
- `static boolean isStrict(int);`

16.5 Uma aplicação

Para ilustrar a aplicação de reflexão computacional, consider o desenvolvimento de uma pequena aplicação que solicita do usuário o nome de uma classe da biblioteca de Java e, a partir do nome informado, recupera o objeto `Class` da classe correspondente e imprime diversas informações sobre as construtoras, métodos e campos dessa classe.

A classe `TestaReflexao` é constituída de quatro métodos privados e o método public `main`, conforme detalhados a seguir.

```
1  import java.lang.reflect.*;
2  import java.io.*;
3  public class TestaReflexao {
4     private PrintStream o = System.out;
5     public  static void main(String[] args) {...}
6     private static void Processa(String nome) {...}
7     private static void imprimaConstrutoras(Class c) {...}
8     private static void imprimaMetodos(Class c) {...}
9     private static void imprimaCampos(Class c) {...}
10 }
```

O método **main** de **TestaReflexao** interage com o usuário, pedindo o nome completo de uma classe, e faz o processamento proposto. O programa encerra-se com a entrada da palavra **"fim"**.

```
1  public static void main(String[] args) {
2     String nome =  " ";
3     BufferedReader infile =
4       new BufferedReader(new InputStreamReader(System.in));
5     while (true) {
6        System.out.print("Nome completo da classe: ");
7        try {
8            nome = infile.readLine();
9            if (!"".equals(nome)) {
10              if ("fim".equals(nome)) System.exit(0);
11              Processa(nome);
12           }
13       }
14       catch (ClassNotFoundException e) {
15              System.out.println("---Nao achou " + nome);}
16       catch(IOException e) {
17              System.out.println("---Erro de leitura"); }
18    }
19 }
```

O método **Processa** recebe o nome da classe a ser pequisada, busca o objeto **Class** que a descreve, determina sua superclasse

e lista em detalhes as suas construtoras, seus métodos e campos, como pode ser visto no seguinte código:

```
1  private static void Processa(String nome)
2                      throws ClassNotFoundException {
3     Class<?> c = Class.forName(nome);
4     Class<?> superc = c.getSuperclass();
5     System.out.print("classe " + nome);
6     if (superc != null && !superc.equals(Object.class))
7          System.out.print(" extends " + superc.getName());
8     System.out.println();
9     imprimaConstrutoras(c);  System.out.println();
10    imprimaMetodos(c);       System.out.println();
11    imprimaCampos(c);        System.out.println("\n}");
12 }
```

O método **imprimaConstrutoras** recebe um objeto **Class** como parâmetro, do qual obtém os detalhes necessários.

```
1  private static void imprimaConstrutoras(Class<?> c){
2     Constructor<?>[] construtoras =
3                              c.getDeclaredConstructors();
4     for (int i = 0; i < construtoras.length; i++) {
5        Constructor<?> cons = construtoras[i];
6        Class<?>[] tiposDosParam = cons.getParameterTypes();
7        String nome = cons.getName();
8        o.print
9            (" " + Modifier.toString(cons.getModifiers()));
10       o.print(" " + nome + "(");
11       for (int j = 0; j < tiposDosParam.length; j++) {
12           if (j > 0) System.out.print(", ");
13           o.print(tiposDosParam[j].getName());
14       }
15       o.println(");");
16    }
17 }
```

Analogamente, o método **imprimaMetodos** exibe os cabeçalhos dos métodos do descritor **Class**.

```
1  private static void imprimaMetodos(Class<?> c) {
2     Method[] metodos = c.getDeclaredMethods();
3     for (int i = 0; i < metodos.length; i++) {
4       Method m = metodos[i];
5       String nome = m.getName();
6       Class<?> tipoRet = m.getReturnType();
7       Class<?>[] tiposDosParam = m.getParameterTypes();
8       o.print
9           ("   " + Modifier.toString(m.getModifiers()));
10      o.print
11          (" " +tipoRet.getName()+ " " +nome+ "(");
12      for (int j = 0; j < tiposDosParam.length; j++) {
13           if (j > 0) System.out.print(", ");
14           System.out.print(tiposDosParam[j].getName());
15      }
16      o.println(");");
17    }
18 }
```

Por fim, o método **imprimaCampos** finaliza o processamento do objeto descritor **Class** passado como parâmetro detalhando as declaração dos campos recuperados de descritor.

```
1  private static void imprimaCampos(Class<?> c) {
2     Field[] campos = c.getDeclaredFields();
3     for (int i = 0; i < campos.length; i++) {
4        Field f = campos[i];
5        Class<?> t  = f.getType();
6        String nome = f.getName();
7        o.print("   " + Modifier.toString(f.getModifiers()));
8        o.println(" " + t.getName() + " " + nome + ";");
9     }
10 }
```

A execução de `ReflectionTest` para a classe `java.lang.Class` produz a seguinte saída[1]:

```
Nome completo da classe ou fim: java.lang.Class
classe java.lang.Class
   private java.lang.Class(java.lang.ClassLoader, java.lang.Class);

   private void checkPackageAccess(java.lang.SecurityManager, java.lang.
     ClassLoader, boolean);
   public static java.lang.Class forName(java.lang.String, boolean,
     java.lang.ClassLoader);
   public static java.lang.Class forName(java.lang.Module,
     java.lang.String);
   public static java.lang.Class forName(java.lang.String);
   private static native java.lang.Class forName0(java.lang.String,
      boolean, java.lang.ClassLoader, java.lang.Class);
   public java.lang.String toString();
   public java.lang.Module getModule();
   public java.security.ProtectionDomain getProtectionDomain();
   public native boolean isAssignableFrom(java.lang.Class);
   public native boolean isInstance(java.lang.Object);
   public native int getModifiers();
   public native boolean isInterface();
   public native boolean isArray();
   public native boolean isPrimitive();
   public native java.lang.Class getSuperclass();
   public java.lang.Object cast(java.lang.Object);
   private static native void registerNatives();
   public java.lang.String getName();
   static java.lang.reflect.Field access$100([Ljava.lang.reflect.Field;,
      java.lang.String);
   private java.lang.Class$ReflectionData reflectionData();
    java.util.Map enumConstantDirectory();
   private java.lang.Class$AnnotationData annotationData();
   public java.lang.String toGenericString();
   public java.lang.Object newInstance();
   public boolean isAnnotation();
   public boolean isSynthetic();
   private native java.lang.String getName0();
   public java.lang.ClassLoader getClassLoader();
    java.lang.ClassLoader getClassLoader0();
   public [Ljava.lang.reflect.TypeVariable; getTypeParameters();
   public java.lang.reflect.Type getGenericSuperclass();
   public java.lang.Package getPackage();
   public java.lang.String getPackageName();
   private [Ljava.lang.Class; getInterfaces(boolean);
   public [Ljava.lang.Class; getInterfaces();
```

[1]Alguma linhas abaixo foram quebradas para ajuste da página

```
private native [Ljava.lang.Class; getInterfaces0();
public [Ljava.lang.reflect.Type; getGenericInterfaces();
public java.lang.Class getComponentType();
public native [Ljava.lang.Object; getSigners();
native void setSigners([Ljava.lang.Object;);
public java.lang.reflect.Method getEnclosingMethod();
private native [Ljava.lang.Object; getEnclosingMethod0();
private java.lang.Class$EnclosingMethodInfo getEnclosingMethodInfo();
private static java.lang.Class toClass(java.lang.reflect.Type);
public java.lang.reflect.Constructor getEnclosingConstructor();
public java.lang.Class getDeclaringClass();
private native java.lang.Class getDeclaringClass0();
public java.lang.Class getEnclosingClass();
public java.lang.String getSimpleName();
public java.lang.String getTypeName();
public java.lang.String getCanonicalName();
public boolean isAnonymousClass();
public boolean isLocalClass();
public boolean isMemberClass();
private java.lang.String getSimpleBinaryName();
private native java.lang.String getSimpleBinaryName0();
private boolean isTopLevelClass();
private boolean isLocalOrAnonymousClass();
private boolean hasEnclosingMethodInfo();
public [Ljava.lang.Class; getClasses();
public [Ljava.lang.reflect.Field; getFields();
public [Ljava.lang.reflect.Method; getMethods();
public [Ljava.lang.reflect.Constructor; getConstructors();
public java.lang.reflect.Field getField(java.lang.String);
public transient java.lang.reflect.Method getMethod(
   java.lang.String, [Ljava.lang.Class;);
public transient java.lang.reflect.Constructor getConstructor(
   [Ljava.lang.Class;);
public [Ljava.lang.Class; getDeclaredClasses();
public [Ljava.lang.reflect.Field; getDeclaredFields();
public [Ljava.lang.reflect.Method; getDeclaredMethods();
public [Ljava.lang.reflect.Constructor; getDeclaredConstructors();
public java.lang.reflect.Field getDeclaredField(java.lang.String);
public transient java.lang.reflect.
   Method getDeclaredMethod(java.lang.String, [Ljava.lang.Class;);
transient java.util.List getDeclaredPublicMethods(
   java.lang.String, [Ljava.lang.Class;);
public transient java.lang.reflect.Constructor getDeclaredConstructor(
   [Ljava.lang.Class;);
public java.io.InputStream getResourceAsStream(java.lang.String);
public java.net.URL getResource(java.lang.String);
private boolean isOpenToCaller(java.lang.String, java.lang.Class);
private native java.security.ProtectionDomain getProtectionDomain0();
static native java.lang.Class getPrimitiveClass(java.lang.String);
```

```
private void checkMemberAccess(
  java.lang.SecurityManager, int, java.lang.Class, boolean);
private java.lang.String resolveName(java.lang.String);
private java.lang.Class$ReflectionData newReflectionData(
  java.lang.ref.SoftReference, int);
private native java.lang.String getGenericSignature0();
private sun.reflect.generics.factory.GenericsFactory getFactory();
private sun.reflect.generics.repository.ClassRepository getGenericInfo();
native [B getRawAnnotations();
native [B getRawTypeAnnotations();
static [B getExecutableTypeAnnotationBytes(java.lang.reflect.Executable);
native jdk.internal.reflect.ConstantPool getConstantPool();
private [Ljava.lang.reflect.Field; privateGetDeclaredFields(boolean);
private [Ljava.lang.reflect.Field; privateGetPublicFields(java.util.Set);
private static void addAll(
  java.util.Collection, [Ljava.lang.reflect.Field;);
private [Ljava.lang.reflect.Constructor;
  privateGetDeclaredConstructors(boolean);
private [Ljava.lang.reflect.Method; privateGetDeclaredMethods(boolean);
private [Ljava.lang.reflect.Method; privateGetPublicMethods();
private static java.lang.reflect.Field searchFields(
  [Ljava.lang.reflect.Field;, java.lang.String);
private java.lang.reflect.Field getField0(java.lang.String);
private static java.lang.reflect.Method searchMethods(
  [Ljava.lang.reflect.Method;, java.lang.String, [Ljava.lang.Class;);
private java.lang.reflect.Method getMethod0(
   java.lang.String, [Ljava.lang.Class;);
private java.lang.PublicMethods$MethodList getMethodsRecursive(
  java.lang.String, [Ljava.lang.Class;, boolean);
private java.lang.reflect.Constructor getConstructor0(
  [Ljava.lang.Class;, int);
private static boolean arrayContentsEq(
  [Ljava.lang.Object;, [Ljava.lang.Object;);
private static [Ljava.lang.reflect.Field;
  copyFields([Ljava.lang.reflect.Field;);
private static [Ljava.lang.reflect.Method;
  copyMethods([Ljava.lang.reflect.Method;);
private static [Ljava.lang.reflect.Constructor;
  copyConstructors([Ljava.lang.reflect.Constructor;);
private native [Ljava.lang.reflect.Field;
  getDeclaredFields0(boolean);
private native [Ljava.lang.reflect.Method;
  getDeclaredMethods0(boolean);
private native [Ljava.lang.reflect.Constructor;
  getDeclaredConstructors0(boolean);
private native [Ljava.lang.Class; getDeclaredClasses0();
private java.lang.String methodToString(
  java.lang.String, [Ljava.lang.Class;);
public boolean desiredAssertionStatus();
```

```
private static native boolean desiredAssertionStatus0(java.lang.Class);
public boolean isEnum();
private static jdk.internal.reflect.ReflectionFactory getReflectionFactory();
public [Ljava.lang.Object; getEnumConstants();
 [Ljava.lang.Object; getEnumConstantsShared();
private java.lang.String cannotCastMsg(java.lang.Object);
public java.lang.Class asSubclass(java.lang.Class);
public java.lang.annotation.Annotation getAnnotation(java.lang.Class);
public boolean isAnnotationPresent(java.lang.Class);
public [Ljava.lang.annotation.Annotation;
  getAnnotationsByType(java.lang.Class);
public [Ljava.lang.annotation.Annotation; getAnnotations();
public java.lang.annotation.Annotation
  getDeclaredAnnotation(java.lang.Class);
public [Ljava.lang.annotation.Annotation;
  getDeclaredAnnotationsByType(java.lang.Class);
public [Ljava.lang.annotation.Annotation; getDeclaredAnnotations();
private java.lang.Class$AnnotationData createAnnotationData(int);
 boolean casAnnotationType(sun.reflect.annotation.AnnotationType,
    sun.reflect.annotation.AnnotationType);
 sun.reflect.annotation.AnnotationType getAnnotationType();
 java.util.Map getDeclaredAnnotationMap();
public java.lang.reflect.AnnotatedType getAnnotatedSuperclass();
public [Ljava.lang.reflect.AnnotatedType; getAnnotatedInterfaces();
static [Ljava.lang.reflect.Field; access$000(java.lang.Class, boolean);

private static final int ANNOTATION;
private static final int ENUM;
private static final int SYNTHETIC;
private transient volatile
  java.lang.reflect.Constructor cachedConstructor;
private transient volatile
  java.lang.Class newInstanceCallerCache;
private transient java.lang.String name;
private transient java.lang.Module module;
private transient java.lang.String packageName;
private final java.lang.Class componentType;
private static java.security.ProtectionDomain allPermDomain;
private transient volatile
  java.lang.ref.SoftReference reflectionData;
private transient volatile int classRedefinedCount;
private transient volatile
  sun.reflect.generics.repository.ClassRepository genericInfo;
private static final [Ljava.lang.Class; EMPTY_CLASS_ARRAY;
private static final long serialVersionUID;
private static final
  [Ljava.io.ObjectStreamField; serialPersistentFields;
private static
  jdk.internal.reflect.ReflectionFactory reflectionFactory;
```

```
    private transient volatile [Ljava.lang.Object; enumConstants;
    private transient volatile java.util.Map enumConstantDirectory;
    private transient volatile
      java.lang.Class$AnnotationData annotationData;
    private transient volatile
      sun.reflect.annotation.AnnotationType annotationType;
    transient java.lang.ClassValue$ClassValueMap classValueMap;
}
```

De fato, a classe `Class` é muito complexa. O que foi apresentado neste capítulo é mera introdução.

16.6 Conclusão

A atividade de se focalizar nos elementos intrínsecos de um programa, solicitar-lhe informações de si e manipulá-las é chamada de **reflexão computacional**.

Há situações que essas operações são úteis como a habilidade de escrever códigos mais genéricos, com a capacidade de recuperar nomes de classes ou outros atributos para fins registros de execução.

Exercícios

1. Qual é a utilidade da reflexão? Identifique situações em que reflexão é indispensável.
2. Quais seriam as desvantagens do uso de reflexão?

Notas bibliográficas

Recomenda-se a leitura do Capítulo 16, Reflexão, do livro de Ken Arnold et alii [2].

Capítulo 17

Linhas de Execução

Uma sequência de instruções que são executadas uma de cada vez forma uma linha de execução ou linha, a qual, tipicamente, executa uma tarefa. Em muitas situações, um sistema consiste em uma única linha, produzindo resultados a partir de entradas fornecidas. Há, contudo, muitas aplicações em que é imperativa a execução em paralelo das tarefas de um programa, como ocorre, por exemplo, em um que suporta o processo de atendimento de usuários de um sistema de vendas pela Internet. Um programa desse tipo deve ser capaz de atender vários clientes simultaneamente, cada um interagindo com ele em diferentes pontos do processo de compra.

Sistemas desse tipo usam técnicas e recursos de Programação Concorrente com o objetivo de garantir acesso sincronizado e exclusivo no tempo a dados comuns e de tornar o atendimento dos usuários mais eficiente do que seria se fosse feito de forma sequencial, um usuário por vez.

Programação concorrente lida com dois tipos de processos: processos leves (*light-weight processes*) e processos pesados (*heavy-weight processes*). A principal diferença entre esses dois tipos de processos é a forma como os recursos usados por cada um são administrados e compartilhados.

Com processos pesados, os espaços de endereçamento são independentes, e dados são compartilhados por meio de protocolos

de comunicação bem definidos, como *sockets* e *pipes*, enquanto os processos leves compartilham diretamente seus espaços de endereçamento de memória de *heap* e de pilha, embora cada um possa ter uma porção própria da pilha de execução, enquanto compartilham outras partes.

Quando há uma troca de controle entre processos pesados, os registros do processo em execução devem ser salvos para posterior retomada de controle, e os registros do processo a assumir o controle devem ser carregados no processador. Com processos leves, grande parte dos registros não precisam ser salvos ou restaurados, pois são por eles compartilhados, sendo tudo executado sob o total controle do programa.

Linhas são processos leves que podem ser criados em qualquer ponto do programa para ser executados concorrentemente com outros processos do mesmo programa. O controle da execução dessas linhas, inclusive das prioridades de cada uma, é estabelecido explicitamente pelo próprio programa.

Cada linha tem uma prioridade de execução, que é um inteiro pertencente ao intervalo definido pelas constantes `MIN_PRIORITY` e `MAX_PRIORITY`. Cada aplicação tem total controle das prioridades de suas linhas, podendo alterá-las a qualquer momento.

Linhas de Java podem ser *user* ou *daemon* e herdam o tipo e prioridade de sua criadora, mas, antes de ser iniciadas, seus tipos podem ser redefinidos para *daemon* via o comando `setDaemon(true)`, ou para *user*, via `setDaemon(false)`.

Quando a JVM é iniciada para uma dada classe, uma linha do tipo *user* com prioridade `NORM_PRIORITY` é automaticamente criada, e o método `main` dessa classe entra em execução. Essa é a linha principal da aplicação.

A execução de um programa, se nele não houver linhas ativas do

tipo *user* em execução, termina quando seu `main` termina. Caso contrário, a aplicação aguarda o término de todas as suas linhas do tipo user. A aplicação não espera por linhas do tipo *daemon*, isto é, todas morrem junto com a aplicação.

17.1 Criação de linhas

Linhas permitem que o programa execute mais de uma de suas tarefas em paralelo. No caso de haver apenas um processador disponível, a execução dessas tarefas é feita de forma entremeada, havendo um compartilhamento alternado do processador.

No caso de se ter mais de um processador, pode-se de fato haver um paralelismo real de execução entre as linhas. Entretanto, o número de processadores disponíveis não deve ser considerado para a correção de um programa multilinha (*multithreading*): todo programa deve codificado para funcionar correta e independentemente do número de processadores, que deve ter apenas efeito seu desempenho.

Linhas são criadas a partir da classe `java.lang.Thread`, a qual disponibiliza um rico repertório de operações para administrá-las, entre as quais há vários tipos de construtoras, operações para iniciar, sincronizar e terminar linhas, controlar prioridades, coletar e configurar dados sobre o processo. Algumas dessas operações da classe `Thread` são:

- `Thread()`:
  constrói uma nova linha, cujo fluxo de execução é dado pelo método `run`.
- `Thread(String nome)`:
  constrói uma nova linha, cujo fluxo de execução é dado pelo método `run` e associa a ela um nome.

- `Thread(Runnable r)`:
  constrói uma nova linha, cujo fluxo de execução é dado pelo método `run` do objeto r.
- `void start()`:
  coloca a linha no estado **executável**, pronta para iniciar a execução do seu método `run()`.
- `void run()`:
  função principal que define o corpo da linha. Não deve ser chamada diretamente pelo usuário. Nada faz.
- `final String getName()`:
  retorna o string identificador da `Thread`.
- `final setName(String nome)`
  `throws securityException`:
  muda o nome identificador da linha.
- `final setDaemon(boolean on)`:
  marca a linha como *daemon*, se `on == true`, senão, *user*.
- `final static int MIN_PRIORITY`:
  prioridade mínima (1) que uma linha pode ter.
- `final static int NORM_PRIORITY`:
  prioridade default de uma linha (5).
- `final static int MAX_PRIORITY`:
  prioridade máxima (10) de uma linha.
- `static Thread currentThread()`:
  retorna referência da linha que executou essa operação.
- `final void setPriority(int p)`
  `throws IllegalArgumentException`:
  define prioridade da linha, sendo `p` um valor no intervalo de prioridades [`MIN_PRIORITY`, `MAX_PRIORITY`].
- `final int getPriority()`:
  informa a prioridade da linha.

- `static void yield()`:
  faz a linha ceder a vez. Uma outra linha executável poderá ser escalada em seguida. O escalador escolhe a linha executável de maior prioridade, possívelmente a própria linha.
- `static void sleep(long t)`
  `throws InterruptException`:
  coloca a linha em execução para *dormir* por `t` ms, e permite que outras linhas tenham a chance de ser executadas.
- `final boolean isAlive()`:
  retorna verdadeiro se a linha tiver sido iniciada e ainda não parou. Retorna falso se a linha for **nova**, i.e., ainda não tornada **executável**, ou se tiver sido encerrada.

Linhas são representadas por meio de objetos da classe `Thread`, e há duas formas de criá-las: na primeira forma, cria-se um objeto do tipo `Thread`, e o corpo da linha é o seu método `run`, e a outra, é via a implementação da interface `Runnable`, e, nesse caso, seu corpo é o `run` anunciado por essa interface.

O método `run` foi concebido para ser usado estritamente dentro do arcabouço de processamento multilinhas. Chamar o método `run` diretamente, como um método ordinário, é sintaticamente possível, mas isso conflita com o seu papel no esquema de programação concorrente de Java, no qual a passagem do controle de execução ao `run` é feita pelo sistema de escalonamento de tarefas embutido na JVM. A operação `t.start()`, onde `t` é um objeto da classe `Thread`, apenas informa ao escalador de tarefas que o `run` da linha `t` está pronto para ser executado. Sua execução iniciar-se-á de acordo com a política de escalonamento que estiver em vigor.

Uma linha termina quando o seu método `run` atinge seu fim ou sua execução for interrompida por ela mesmo, por exemplo, em atendimento a uma mensagem gerada pela operação `interrupt`.

17.1.1 Criação de linhas via `Thread`

Um caminho para criar uma linha de execução inicia-se pela declaração de uma classe que estende **`Thread`**, sobrepondo seu método **`run`**, como no seguinte código:

```
1 class T extends Thread {
2    public void run() { ... corpo da thread ... }
3 }
```

Em seguida, aloca-se um objeto dessa classe que estende **`Thread`**, e anuncia, via operação **`start()`**, que a linha está pronta para ser executada, como sugere o programa `B` abaixo, que cria duas linhas de execução `t1` e `t2`, dispara a execução de seus respectivos métodos **`run`** e depois executa outras tarefas.

```
1  public class B {
2    public static void main(String[] a) {
3      T t1 = new T();  // cria linha 1
4      T t2 = new T();  // cria linha 2
5      ...              // configura as linhas adequadamente
6      t1.start();      // dispara linha1
7      t2.start();      // dispara linha 2
8      ...              // outras tarefas
9    }
10 }
```

Os comandos 3 e 4 do método **`main`** de `B` acima criam as linhas `t1` e `t2`, mas não iniciam sua execução. Nesse ponto, os objetos das linhas criadas podem ser configurados via as operações disponibilizadas pela classe **`Thread`**.

Os comandos 6 e 7 disparam a execução dos respectivos métodos **`run`** das linhas `t1` e `t2`, mas isso ainda não inicia a execução de seus métodos **`run`**.

O disparo de uma linha não significa que ela iniciará imediatamente a execução de seu **run**, porque o efeito do disparo é apenas o de colocá-la na fila de processos prontos para execução. A execução de uma linha somente é de fato iniciada quando sua vez chegar, de acordo com a política de prioridade do ambiente de execução.

Para ilustrar o presente método de criação de linhas de execução, considere o programa **TestaGentil** abaixo, que dispara duas linhas, as quais imprimem seus nomes **n** vezes.

As linhas disparadas são da classe **Gentil**, assim chamada porque suas linhas são *gentis* no sentido de que não monopolizam o processador: após cada **f** impressões, uma oferece à outra a chance de assumir o controle do processador.

Os valores de **n** e **f** são informados no ato de criação de cada linha. E as operações **getName** e **yield** são herdadas de **Thread**.

```
1  class Gentil extends Thread {
2    private int f, n;
3    Gentil(int f, int n) {this.f = f; this.n = n;}
4    public void run() {
5       for (int i=1; i<=n; i++) {
6               System.out.print(getName() + " ");
7               if (i%f == 0) Thread.yield();
8       }
9       System.out.print("\n" + getName() + " terminou");
10   }
11 }
12 public class TestaGentil {
13    public static void main(String[] args) {
14       Thread t1 = new Gentil(3,20);
15       Thread t2 = new Gentil(2,10);
16       t1.start();  t2.start();
17       System.out.println("\nMain chegou ao fim");
18   }
19 }
```

O programa `TestaGentil` produz a seguinte saída:

```
Main chegou ao fim
Thread-1 Thread-1 Thread-0 Thread-0 Thread-0 Thread-1
Thread-1 Thread-0 Thread-0 Thread-0 Thread-1 Thread-1
Thread-0 Thread-0 Thread-0 Thread-1 Thread-1 Thread-0
Thread-0 Thread-0 Thread-1 Thread-1 Thread-0 Thread-0
Thread-0 Thread-0 Thread-0 Thread-0 Thread-0 Thread-0
Thread-0 terminou
Thread-1 terminou
```

Na qual pode-se observar que o método `main` ficou aguardando o fim das linhas por ele criadas para só então finalizar sua própria execução.

17.1.2 Criação de linhas via `Runnable`

O método de criação de linha por extensão de `Thread` é simples, mas tem o inconveniente de a classe que estende `Thread` não poder simultaneamente estender outra classe, o que dificulta reúso de código. Assim, para contornar essa dificuldade, recomenda-se um método alternativo, que faz uso da interface

```
public interface Runnable {
   void run();
}
```

disponível na biblioteca `java.lang`.

O primeiro passo é implementar a interface `Runnable`, podendo-se estender alguma outra classe de interesse para fins de reúso de código, e definir o método `run()`, que será o corpo da linha, o qual será executado em algum momento após ser disparado, como esquematizado abaixo:

```
class R extends X implements Runnable {
   public void run() { ... }
}
```

Note que objetos de uma classe que implementa **Runnable** não é uma linha, pois linhas têm que ser objetos da classe **Thread**. Assim, para tornar o **run** da classe que implementa **Runnable** efetivamente o corpo de uma linha, é necessário alocar um objeto do tipo **Thread**, informando via sua construtora, que o método **run** a ser usado é o da classe que implementou a interface **Runnable** e não o **run** definido em **Thread**. Isso é obtido por um comando do tipo

```
Thread linha = new Thread(new R());
```

onde **R** é a classe que implementa **Runnable**.

Quando se criam linhas por extensão direta de **Thread**, a construtora **Thread()**, que é automaticamente chamada pela construtora de classe que estendeu **Thread**, aloca uma linha cujo **run** é o definido nessa classe. Por outro lado, na criação de linhas via **Runnable**, a construtora **Thread(Runnable r)** deve ser usada para construir uma nova linha na qual o fluxo de execução é o **run** do objeto **r** que lhe foi passado como parâmetro.

O programa abaixo exemplifica esse método:

```
1  public class TestaRunnable {
2     public static void main(String[] args) {
3        Thread linha1 = new Thread(new R()); //cria linha 1
4        Thread linha2 = new Thread(new R()); //cria linha 2
5        ... // configura linhas adequadamente
6        linha1.start();  // inicia linha 1
7        linha2.start();  // inicia linha 2
8     }
9  }
10 class R extends X implements Runnable {
11    public void run() { ... }
12 }
```

As operações acima são suficientes para criar novas linhas, entretanto, observe que a classe que estende a interface **Runnable**

não herda as operações definidas na classe **Thread**, como mostram os exemplos a seguir.

Considere a classe **A** definida como uma subclasse de **Thread** e que faz uso dos serviços **getName**, **getPriority**, **setPriority** e **sleep**, todos providos pela classe **Thread**, conforme o código abaixo:

```
1 class A extends Thread {
2    public void run() {
3      System.out.println("Extensão de Thread " + getName());
4      setPriority(getPriority() + 1);
5      try {Thread.sleep(1000);}
6      catch (InterruptedException e) {return;}
7    }
8 }
```

O programa **ExemploI**, que usa a classe **A**, cria duas linhas de execução baseadas em **A** e as dispara imediatamente.

```
1 public class ExemploI {
2    public static void main(String[] a) {
3       A t1 = new A();
4       t1.start();
5       A t2 = new A();
6       t2.start();
7       Thread t0 = Thread.currentThread();
8       System.out.println("Main - " + t0.getName());
9    }
10 }
```

O resultado impresso por **ExemploI** é:

```
Extensao de Thread Thread-1
Main - main
Extensao de Thread Thread-0
```

Observe que o efeito dos comandos nas linhas 7 e 8 de **ExemploI** é capturar o objeto descritor da linha em execução e imprimir seu nome identificador, no caso **main**.

O método alternativo de construir linhas, via implementação da interface **Runnable**, segue o mesmo estilo, mas agora o programa principal tem o seguinte formato:

```
1  public class ExemploII {
2     public static void main(String[] a) {
3        Thread t1 = new Thread(new B());
4        t1.start();
5        Thread t2 = new Thread(new B());
6        t2.start();
7        Thread t0 = Thread.currentThread();
8        System.out.println("Main - " + t0.getName());
9     }
10 }
```

onde o comando **new Thread(new B())** foi usado duas vezes para criar as linhas baseadas em **B**.

Para fins de comparação dos métodos de criação de linhas, considere a classe **B** abaixo, que implementa **Runnable** e tem um **run** que executa as mesmas ações definidas no **run** da classe **A**.

```
1  class B implements Runnable {
2     public void run() {
3       System.out.println("Impl de Runnable " + getName());
4       setPriority( getPriority() +1 );
5       try {Thread.sleep(1000);}
6       catch (InterruptedException e) {return;}
7     }
8  }
```

O código acima não funciona, pois contém erros devido às tentativas de uso dos serviços **getName**, **getPriority** e **setPriority** da classe **Thread**, e, que estão fora do escopo da classe **B**, a qual não tem relação alguma de parentesco com **Thread**.

Para resolver esse problema, é preciso trazer o acesso aos serviços de **Thread** para o escopo de **B**, o que pode ser feito via a operação

```
Thread t = Thread.currentThread();
```

que captura a referência ao objeto descritor da linha que executou esse comando, como mostrado na nova versão de `B`, apresentada abaixo, onde usa-se `t` para qualificar os serviços desejados de `Thread`.

A nova versão de `B`, que agora funciona, é a seguinte:

```
1 class B implements Runnable {
2   public void run() {
3     Thread t = Thread.currentThread();
4     System.out.println("Impl de Runnable " + t.getName());
5     t.setPriority(t.getPriority() +1 );
6     try {Thread.sleep(1000);}
7     catch (InterruptedException ie) {return;}
8   }
9 }
```

E o resultado do programa `ExemploII` é:

```
Impl de Runnable Thread-0
Impl de Runnable Thread-1
Main - main
```

diferenciando do resultado de `ExemploI` apenas na ordem de impressão dos resultados, o que é de se esperar.

17.1.3 Ciclo de vida de linhas

Diferentemente de processos pesados, que são em geral entidades independentes, linhas são componentes de um mesmo programa totalmente sob seu controle, podendo ser iniciadas, bloqueadas ou finalizadas de acordo com o interesse da aplicação.

Resumidamente, os principais atos associados a linhas envolvem as seguintes operações:

- uma linha `t` do tipo `T` pode ser criada com a operação do tipo `T t = new T();`

- a linha `t` é iniciada com a chamada `t.start()`, que estabelece que sua execução deve iniciar-se pelo método `run()` associado;
- toda linha termina com o fim de seu `run()`, exceto a principal, que se encerra com o fim de seu método `main`;
- uma linha pode solicitar a interrupção de uma outra linha `t` enviando-lhe uma exceção via o comando `t.interrupt()`. Essa exceção denominada `InterruptException` sinaliza o pedido à linha `t`, que pode então decidir por encerrar seu processamento;
- toda linha está sujeita à **preempção** por uma outra linha de maior prioridade;
- uma linha pode, via `this.yield()`, ceder o controle do processador para uma outra assumir o controle;
- qualquer linha pode ajustar sua prioridade ou as de outras linhas, com valores que vão de 1 (baixa) a 10 (alta), por meio das operações `getPriority` e `setPriority`;
- cada linha possui uma pilha tamanho padrão de 400kb aproximadamente, mas esse tamanho pode ser reconfigurado.

Versões anteriores de Java ofereciam as operações `t.stop()`, para encerrar a linha `t`, `t.suspende()`, para interromper temporariamente a execução da linha `t` e `t.resume()`, para retomar a execução interrompida de `t`. Essas operações foram declaradas **depreciadas** em edições posteriores de Java, e, não devem mais ser usadas, e, por isso, não são aqui tratadas.

Uma linha pode estar em um dos estados: **novo**, **executável**, **bloqueado** ou **encerrado**, definidos da seguinte forma:

- **novo**:
  é o estado inicial da linha logo após a sua criação.
- **executável**:
  é o estado de uma linha após a execução do método `start()`.

Nesse estado, a linha pode estar *rodando* ou não. Em alguns sistemas, uma linha executa até que outra de maior prioridade tome-lhe o controle. Em outros, usa-se a política de tempo compartilhado.

- **bloqueado**:
  é o estado de uma linha que para ela for chamado o método `sleep()`, o método `wait()` ou que iniciou uma operação de entrada/saída.
- **encerrado**:
  é o estado de uma linha encerrada porque seu método `run()` terminou.

Quando uma linha bloqueada é reativada, o escalonador somente lhe dá o controle imediatamente se ela tiver prioridade mais alta do que a atualmente em execução. Se esse não é o caso, a linha terá que aguardar sua vez na fila de linhas executáveis. A passagem de estado **bloqueado** para **executável** ocorre quando:

- a linha que *dormia* completou o tempo `ms` do `sleep(ms)`;
- a linha que estava bloqueada devido a um `wait(...)`, e um `notify` ou `notifyAll` foi enviado ao objeto controlador do bloqueio ou
- a operação de E/S que a linha requisitou terminou.

A reativação de uma linha somente é efetiva se combinar com a razão que a bloqueou: uma linha que estiver bloqueada por E/S somente será reativada se a operação de E/S solicitada for concluída. Tentativas de ativação de uma linha incompatível com o seu estado levanta a exceção `IllegalThreadStateException`.

Toda linha recebe a prioridade de seu criador. Pode-se alterar sua prioridade com a operação `setPriority`. A definição da prioridade adequada é importante, porque o escalador de linhas

sempre escolhe a de maior prioridade e usa alguma política que não faz parte da linguagem Java para decidir o que fazer quando houver mais de uma linha executável com a mesma prioridade.

O sistema de execução Java pode suspender a execução de uma linha de maior prioridade para que outras tenham a chance, mas não se tem garantia de quando isto ocorrerá. Tipicamente, a linha executável continua sendo executada até que um dos seguintes eventos ocorra: ela cede a vez via `yield`, executa uma operação `sleep`, deixa de ser executável, é bloqueada ou é substituída por uma linha de maior prioridade.

17.2 Sincronização de linhas

Linhas de um mesmo programa geralmente compartilham áreas de dados para escrita e leitura de informações. O paralelismo, ainda que entremeado, na execução dessas linhas requer que as operações de escrita e leitura de dados compartilhados sejam devidamente sincronizadas no tempo de forma a evitar sua corrupção, o que pode ocorrer devido ao fenômeno conhecido por *race condition*.

A sincronização dessas operações requer que seus trechos de código sejam identificados e reunidos em grupos para garantir exclusividade no tempo. Isso significa que quando uma linha estiver executando um desses trechos, nenhuma outra poderá simultaneamente executar trechos de código do mesmo grupo. Esses trechos de código são denominados **regiões críticas** e formam grupos vinculados a objetos sincronizadores. Java provê quatro recursos para garantir execução de regiões críticas de forma exclusiva: comando `synchronized`, métodos `synchronized`, métodos `synchronized` estáticos e objetos monitores.

17.2.1 Comandos sincronizados

Comandos sincronizados são um mecanismo destinado a criar grupos de regiões críticas. Em um programa pode haver vários grupos de regiões críticas, onde cada grupo sincroniza no tempo a execução de seus constituintes. Uma forma de organizar esses agrupamentos é por meio do comando twsynchronized.

O comando `synchronized(obj1){cmds1}`, no qual `obj1` é uma referência a um objeto, vincula a autorização de execução dos comandos da região crítica `cmds1` ao objeto apontado por `obj1`. Um outro comando da forma `synchronized(obj2){cmds2}` em que `obj2` referencia o mesmo objeto que `obj1` inclui o `cmds2` no mesmo grupo de regiões críticas de `cmds1`. Em um programa pode haver vários comandos `synchronized` com essas características, formando um grupo com várias regiões críticas, todas sincronizadas pelo mesmo objeto apontado, o qual pode ser bloqueado e desbloqueado no processo de execução dos comandos `synchronized`.

Toda vez que um comando `synchronized(obj){cmds}` é atingido no fluxo de execução de uma linha, e o objeto referenciado por `obj` estiver desbloqueado, a linha adquire o seu bloqueio imediatamente e passa a executar os comandos em `cmds` de forma exclusiva. O bloqueio de `obj` perdura até o término de `cmds`, podendo ser temporariamente suspenso pela operação `wait` da classe `Object`, conforme descrito na Seção 17.3.

Por outro lado, enquanto o `obj` do comando `synchronized` estiver bloqueado, a linha que tentar executá-lo é colocada em uma fila de espera associada ao objeto sincronizador, e a sua execução somente será retomada após o desbloqueio desse sincronizador.

Todo objeto em Java tem uma fila de sincronização associada para uso do comando `synchronized`. Qualquer objeto pode então ser usado como objeto sincronizador.

Sempre que houver um desbloqueio do objeto sicronizador, uma linha dessa fila é escolhida para retomar sua execução. Se houver mais de uma linha nessa fila no momento do desbloqueio, a escolha é feita conforme as prioridades das candidatas.

Uma linha somente libera o bloqueio do objeto sincronizador por ela adquirido nas seguintes situações:

- a execução da região crítica chegou ao seu fim;
- uma exceção ocorreu e não foi tratada, sendo propagada para fora da região crítica ou
- uma pausa foi solicitada via com comando `wait`.

Operações como `sleep` e `yield` suspendem momentaneamente a execução de linhas, permitindo que outras linhas sejam executadas, mas nunca liberam o bloqueio de objetos sincronizadores das regiões críticas onde estão.

Para ilustrar o uso do comando `synchronized`, considere as classes `L1` e `L2` que executam as regiões críticas marcada por `r1` e `r2` nos trechos de programa abaixo e que são sincronizadas por objetos referenciados por `lock`.

```
1  class L1 extends Thread {
2    Object lock;
3    public L1(Object obj) {lock = obj;}
4    public void run() {... synchronized(lock) {...r1...} ...}
5  }
6
7  class L2 extends Thread {
8    Object lock;
9    public L2(Object obj) {lock = obj;}
10   public void run() {... synchronized(lock) {...r2...} ...}
11 }
```

O programa que cria e dispara linhas baseadas em `L1` e `L2` é o definido pelo `main` da classe `M` abaixo, que gera a alocação de ob-

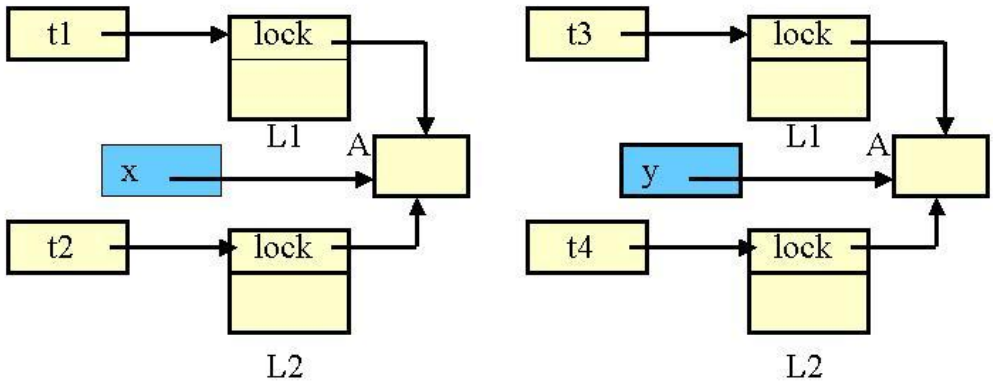


Figura 17.1 Objetos Sincronizadores

jetos apresentada na Fig. 17.1, e opera com dois grupos de regiões críticas, cada um de forma sincronizada no tempo.

```
1 class M {
2   public static void main(String[] args) {
3     A x = new A(); A y = new A();
4     L1 t1 = new L1(x); L1 t3 = new L1(y);
5     L2 t2 = new L2(x); L2 t4 = new L2(y);
6     t1.start(); t2.start(); t3.start(); t4.start();
7   }
8 }
```

Observe que as regiões `r1` e `r2` das linhas `t1` e `t2` são sincronizadas entre si, via `x`, enquanto que as mesmas regiões que são relativas a `t3` e `t4` pertencem ao grupo de sincronização de `y`.

A classe `A` dos objetos sincronizadores usados no exempo é uma classe qualquer, podendo ser a própria classe `Object`.

17.2.2 Controle de término

As seguintes operações da classe `Thread` permite uma linha aguardar o término de outras para progredir em seu fluxo de execução:

- `final void join() throws InterruptedException`:
  linha corrente entra em espera até que a linha receptora do `join` encerre sua execução.
- `final void join(long millis)`
  `throws InterruptedException`:
  linha corrente espera até que a linha encerre sua execução dentro de `millis` milissegundos. Finalizado esse tempo, a linha retoma sua execução. Se `milles == 0`, a espera é por tempo indefinido.
- `final void join(long millis, int nanos)`
  `throws InterruptedException`:
  similar ao método anterior, o tempo de espera tem a precisão de nanossegundos: `millis*1000 + nanos` (nanossegundos). E o tempo `0` também significa espera por tempo indefinido.
- `boolean isAlive()`:
  informa se a linha receptora está viva ou não.

17.2.3 Controle de uma caldeira

Considere a implementação de um sistema de controle de uma caldeira, dotada de dois aparatos físico-químicos responsáveis por incrementar a sua pressão até um certo limite, acima do qual a caldeira pode explodir.

Esses dois aparatos funcionam continuamente em paraledo e, por isso, devem ser implementados por duas linhas, que fazem acesso a um mesmo manômetro.

Essas linhas são independentes entre si quanto aos seus papéis no funcionamento da caldeira, mas para simplificar o exemplo, elas serão aqui implementadas por métodos `run` similares.

O manômetro pode ser modelado pela seguinte classe, que apenas permite incrementar o valor da pressão e informar seu o valor.

```
1  class Manômetro {
2     private int pressao, limite;
3     public Manometro(int limite) {
4        this.limite = limite;
5     }
6     public int incrementa(int incremento) {
7        pressao += incremento;
8        return pressao;
9     }
10    public int pressao() {return pressao;}
11    public int limite() {return limite;}
12 }
```

Um dos aparatos é a linha `p1`, que é um objeto da classe `Aparato1`, e tem a função de incrementar a pressão da caldeira.

```
1  class Aparato1 extends Thread {
2     Manômetro m;
3     Object sincronizador;
4     Aparato1(Manômetro m, Object sincronizador) {
5        this.m = m; this.sincronizador = sincronizador;
6     }
7     public void run() {
8      if (sincronizador!=null) synchronized(sincronizador) {
9          while (m.pressao() <= m.limite()-1) {
10            try {Thread.sleep(10);} catch(Exception e) {};
11            m.incrementa(1);
12         }
13     }
14     else while (m.pressao() <= m.limite()-1) {
15            try {Thread.sleep(10);} catch(Exception e) {};
16            m.incrementa(1);
17     }
18    }
19 }
```

A construtora da classe da linha `p1` recebe os endereços do objeto manômetro e do objeto de sincronização de acesso aos registros desse manômetro. O endereço do objeto `Manômetro` deve ser sempre válido, mas o do sincronizador pode ser nulo.

O método `run` da classe `Aparato1` foi programado para realizar a operação de atualização da pressão registrada no manômetro de forma exclusiva somente quando o objeto `sinalizador` informado à linha tiver valor diferente de `null`.

Certamente, essa ideia de decidir em tempo de execução por usar ou não sincronização no acesso a região crítica não é de uso comum, mas isto foi feito no presente exemplo para enfatizar a necessidade de sincronização quando pode ocorrer o chamado *race condition*, isto é, possibilidade de acessos competitivos a dados comuns por mais de uma linha.

Observe que o `sincronizador` passado à função construtora de `Aparato1` é usado no comando `synchronized` da linha 8 da classe `Aparato1` para controlar a exclusividade temporal na execução do incremento da pressão do manômetro.

Observe também que, na linha 9, o `run` de `Aparato1` faz uma chamada ao serviço `Thread.sleep(10)`, que tem o efeito de liberar o controle do processor e ceder a vez a outra linha em execução. Assim, após decorridos `10` milissegundos, a linha que foi suspensa é devolvida à fila de linhas executáveis e prontas para retomar a execução na linha 11.

Com a execução de `sleep`, a linha não perde a posse do bloqueio da região crítica controlada pelo comando `synchronized` em execução: apenas dá chance para alguma linha progredir na execução.

A implementação do `Aparato2` é quase idêntica a de `Aparato1`, exceto pelo valor do incremento aplicado à pressão da caldeira.

```
class Aparato2 extends Thread {
   Manômetro m;
   Object sincronizador;
   Aparato2(Manômetro m, Object sincronizador) {
      this.m = m; this.sincronizador = sincronizador;
   }
   public void run() {
     if (sincronizador!=null) synchronized(sincronizador) {
         while (m.pressao() <= m.limite()-2) {
           try {Thread.sleep(10);}catch (Exception e) {};
           m.incrementa(2);
         }
     }
     else while (m.pressao() <=m.limite()-2) {
            try {Thread.sleep(10);} catch(Exception e) {};
            m.incrementa(2);
     }
   }
}
```

O `main` do programa `CaldeiraA` abaixo imprime o valor final da pressão quando as linhas `p1` e `p2` houverem terminado.

```
public class CaldeiraA {
   public static void main(String[] args) {
      Manômetro m = new Manômetro(20);
      Object sincronizador = null;
      Aparato1 p1 = new Aparato1(m,sincronizador);
      Aparato2 p2 = new Aparato2(m,sincronizador);
      p1.start(); p2.start();
      try {p1.join();} catch(InterruptedException e) {}
      try {p2.join();} catch(InterruptedException e) {}
      System.out.println("Manometro:" + m.pressao());
  }
}
```

Programa `CaldeiraA`, quando executado várias vezes, ora imprime `Caldeira:20` e ora, `Caldeira:21`, claramente podendo provocar uma explosão. A razão desse mau funcionamento deve-se ao fato de o objeto `sinalizador` ter sido inicializado, na linha 4, com o valor `null` e passado às construtoras de `Aparato1` e `Aparato2`.

O programa `CaldeiraB` apresentado a seguir corrige esse erro, alocando um objeto sincronizador, no comando da linha 4, o qual também é repassado às construtoras de `Aparato1` e `Aparato2`.

```
1  public class CaldeiraB {
2     public static void main(String[] args) {
3        Manômetro m = new Manômetro(20);
4        Object sincronizador = new Object();
5        Aparato1 p1 = new Aparato1(m,sincronizador);
6        Aparato2 p2 = new Aparato2(m,sincronizador);
7        p1.start();
8        p2.start();
9        tryp1.join();catch(InterruptedException e) {}
10       tryp2.join();catch(InterruptedException e) {}
11       System.out.println("Manometro:" + m.pressao());
12    }
13 }
```

Nesse caso, as linhas `p1` e `p2` realizam os devidos incrementos de pressão dentro de regiões críticas. E o resultado é que o programa `CaldeiraB`, quando executado várias vezes, imprime sempre `Caldeira:20`, que é consistente com o esperado.

17.2.4 Métodos sincronizados

Em geral, por razões de desempenho, regiões críticas devem consistir em poucos comandos, o que pode ser obtido pelo uso do comando `synchronized` em várias partes de um método, criando pequenas regiões críticas.

Entretanto, quando todo o corpo de um método é uma região crítica a ser protegida por um objeto sincronizador, pode-se evitar o uso de comandos `synchronized`, apenas acrescentando a palavra `synchronized` ao cabeçalho do método, gerando os métodos sincronizados, que podem ser estáticos ou não-estáticos.

Um método sincronizado não-estático tem a forma

```
synchronized void m() { corpo }
```

que é uma abreviatura de

```
void m() {
   synchronized(this) { corpo }
}
```

E um método sincronizado estático tem a forma:

```
static synchronized void s() { corpo }
```

que é o mesmo que

```
static void s() {
   private static Object sync = A.class;
   synchronized(sync) { corpo }
}
```

onde `A` é a classe que contém o método estático `s`.

Observe que, no primeiro caso, o sincronizador é o objeto `this` para o qual o método foi ativado. E, no caso de métodos estáticos, o sincronizador é o objeto do tipo `Class` descritor da classe que contém o método, sendo portanto o mesmo para todos os métodos `static synchronized` de cada classe.

As regras básicas de funcionamento de métodos sincronizados são:

- métodos `sincronizados` de um mesmo objeto são mutuamente exclusivos;
- método `sincronizados` de objetos distintos, mesmo que da mesma classe, não são mutuamente exclusivos;

- métodos estáticos sincronizados de uma mesma classe são mutuamente exclusivos;
- métodos não-estáticos `sincronizados` não excluem métodos não-sincronizados nem métodos estáticos (sincronizados ou não);
- métodos sincronizados não-estáticos podem chamar sem bloqueio qualquer outro método do mesmo `this`, inclusive os sincronizados não-estáticos da classe, pois o bloqueio pertence à linha, mas chamadas a sincronizados estáticos dependem do bloqueio destes métodos;
- um método sincronizado estático somente pode chamar, sem bloqueio, método estático da classe;
- a redefinição, em subclasses, de um método sincronizado pode ou não ser `synchronized`;
- `super.method()` pode ser sincronizado mesmo se `method()` não o for;
- se um método não-sincronizado chama seu `super` sincronizado, durante a execução do `super()` bloqueia-se o objeto.

Como métodos sincronizados podem chamar métodos não-sincronizados, inclusive de outros objetos, pode ser necessário inspecionar se uma dada linha detém ou não um dado objeto sincronizador. Essa inspeção pode ser feita pelo seguinte método da classe `Thread`:

- `static boolean holdsLock(Object obj)`:
  retorna verdadeiro se e somente se a linha corrente detém o bloqueio do objeto `obj`.

Objetos que têm métodos sincronizados ou métodos que contêm comandos `synchronized` possuem estruturas de dados especiais, que permitem implementar linhas dotadas de mecanismos de exclusão mútua e de cooperação entre si. Esses objetos são denominados **objetos monitores**.

17.3 Objetos monitores

A classe `Object` disponibiliza para implementar objetos monitores os métodos `wait`, `notify` e `notifyAll`, os quais possuem uma fila associada para armazenar descritores de linhas em estado de espera devido à execução da operação `wait()`.

A operação `wait` deve ser usada por uma linha quando os recursos por ela demandados não estiverem disponíveis e ela deseja aguardar o seu provimento. Uma linha que executa um `x.wait()` que esteja dentro de um comando `synchronized` ou do corpo de um método sincronizado, ambos protegidos por um objeto sincronizador `y`, vai para a fila de espera de `x`, e sua execução é interrompida, e o bloqueio administrado por `y` é liberado. Quando a linha interrompida for, mais tarde, reativada, ela recupera seu controle de execução no ponto imediatamente após o `x.wait()`.

A operação `t.notify` é usada para sinalizar ao objeto monitor `t` que algum recurso foi providenciado e talvez a linha na frente de sua fila de espera queira utilizá-lo. As operações `wait` e `notify` são relacionadas quando são aplicadas ao mesmo objeto monitor, isto é, `a.notify()` notifica apenas uma das linhas que executaram um `b.wait()`, se `a` e `b` denotarem o mesmo objeto monitor.

A operação `x.notify()` pode ser ativada de qualquer lugar, dentro ou fora de regiões críticas. Seu efeito é o de acordar apenas uma linha da fila de espera do objeto monitor `x`, isto é, o `notify` remove uma das linhas da fila de espera de `x` e a coloca na fila de linhas prontas para continuar a execução.

É preciso cuidado no uso dessa operação, pois não há garantia de que os recursos esperados pela linha selecionada estejam disponíveis. Isto é, a notificação pode ter sido enviada para a linha errada. E, em consequência, ela terá que retornar à fila de espera, e, o que é mais grave, a linha que deveria ter sido notificada poderá

nunca mais receber qualquer notificação.

Por evitar esse risco, recomenda-se sempre acordar todas as linhas na fila de espera de um objeto `x` usando `x.notifyAll()`, que notifica todas as linhas que estão bloqueadas por terem executado um comando `x.wait`, e todas elas podem, eventualmente, retomar a execução no ponto em que foram bloqueadas.

As operações de classe `Object` relacionadas à sincronização são:

- `final void wait(long timeout) throws InterruptedException`: a linha que emitir essa operação espera ser notificada, via o objeto corrente, mas espera no máximo o tempo especificado `timeout` (ms). Se `timeout==0` espera indefinidamente.
- `final void wait(long mili, int nanos) throws InterruptedException`: como a operação anterior, mas com tempo de espera com precisão de nanossegundos é `mili*1000 + nanos`.
- `final void wait() throws InterruptedException`: o mesmo que `wait(0)`.
- `public final void notify()`: notifica exatamente uma das linhas que esperam por uma condição na fila do objeto monitor corrente do `notify`. Não é possível escolher quem será notificado. Somente a primeira da fila recebe a notificação. Use com cuidado.
- `public final void notifyAll()`: notifica todas as linhas que esperam por alguma condição: todas as linha na fila do objeto são colocadas na filas das linhas prontas para execução.

Há um padrão recorrente de estruturação do código para o uso das operações `wait` e `notify`: a linha que demanda recursos produzidos por outra deve testar uma condícão que revela se esses

recursos estão disponíveis. Isso pode ser feito segundo o seguinte padrão de programação:

```
1 synchronized void execQuandoPuder() {
2     ...
3     while(!condição) {
4         x.wait();
5     };
6     ... // os recursos desejados estão disponíveis
7 }
```

O comando `x.wait()` da linha 4 é ativado quando a expressão lógica `condição` for `false`, e, quando a linha retomar o controle no ponto 5, a condição do comando `while` é reavaliada para verificar se de fato os recursos necessários tornaram-se disponíveis. Se isso não ocorrer, a linha deve executar o `wait` novamente.

Os recursos demandados por uma linha como a descrita acima são providos por outras linhas, que tipicamente, sinalizam a disponibilidade desses recursos executando a operação `x.notify()` ou `x.notifyAll()` para o mesmo objeto monitor apontado por `x` que foi usado na emissão de `x.wait()`. Entretanto, a linha que gera o recurso não tem controle sobre qual linha irá utilizá-lo.

17.3.1 Produtores e consumidores

O problema clássico do Consumidor e Produtor que se comunicam via um *buffer*, apresentado a seguir, ilustra o uso de objetos monitores e das operações `wait` e `notify`.

Nessa implementação, produtores continuamente depositam, um a um, inteiros em um *buffer*, enquanto consumidores retiram continuamente inteiros do mesmo *buffer*, o qual encapsula uma estrutura de dados compartilhada pelos consumidores e produtores, devendo por isso ser cuidadosamente implementado para evitar a

ocorrência da chamada *race condition*.

```
1  class Buffer {
2     private int max;         // capacidade do buffer
3     private int[] valor;    // valores armazenados
4     private int entrada;     // posição de inserção
5     private int saida;       // posição de retirada
6     private int total = 0;  // número de valores no buffer
7     Buffer(int tamanho) {
8         max = tamanho;  valor = new int[max];
9     }
10    public synchronized void insira(int item) { ... }
11    public synchronized int  remova() { ... }
12 }
```

A classe **Buffer** disponibiza uma construtora que aloca a área de armazenamento de inteiros e as operações **insira** e **remova**.

Os métodos **insira** e **remova** foram declarados **synchronized** para que sejam executados de forma exclusiva no tempo, ou seja, se um deles estiver com o controle, as linhas que quiserem executá-los para o mesmo objeto do tipo **Buffer** terão que aguardar sua vez. O método sincronizado **insira** da classe **Buffer** é o seguinte:

```
1  public synchronized void insira(int item) {
2      while(total >= max)
3         try {wait();} catch(InterruptedException e) { };
4      valor[entrada] = item;
5      System.out.println("Inserido no Buffer " + item);
6      if(++entrada == max) entrada = 0;
7      total++;
8      notify();
9  }
```

A primeira ação de **insira** é verificar se há espaço no *buffer* para inserção de um inteiro. Se houver espaço no *buffer*, **insira** procede com as acões para inserção do **item**. Caso não haja, a

linha que acionou essa operação emite, na linha 3, o comando `this.wait()`, que a coloca na fila de espera do objeto monitor `this` e ao mesmo tempo desbloqueia esse objeto para que outra linha possa prosseguir na execução de uma região crítica do grupo do `this`.

A linha colocada em espera, quando retomar a execução, graças a algum `notify` emitido por uma outra linha, fará a reverificação da existência de espaço para inserção de um inteiro no *buffer*, e somente prosseguirá na execução se houver essa garantia. Esse cuidado é indispensável, porque o `notify` que causa a devolução do controle de execuçao à linha não garante que os recursos por ela demandado estejam agora de fato disponíveis. O último ato de `insira` é gerar um `notify` para que alguma linha no momento bloqueada por condição de *buffer* vazio tenha a chance de retomar o controle de execução.

A operação `remova`, a seguir, que também é sincronizada, tem um código complementar ao de `insira`.

```
1  public synchronized int remova() {
2     int temp;
3     while(total<=0)
4        try {wait();} catch(InterruptedException e) { };
5     temp = valor[saida];
6     System.out.println("Removido do Buffer " + temp);
7     if(++saida == max) saida = 0;
8     total--;
9     notify();
10    return temp;
11 }
```

A primeira ação de `remova` é verificar se há no *buffer* item a ser removido. Em caso negativo, um `wait` é executado para que a linha aguarde um `notify` para continuar. O último ato de `remova` é um `notify` para sinalizar espaço livre.

As linhas da classe **Consumidor** têm o método **run** que solicita a remoção de inteiros do *buffer*.

```
1  class Consumidor extends Thread {
2     private Buffer b; private String id;
3     public Consumidor(String id, Buffer b) {
4        this.b = b; this.id = id;
5     }
6     public void run() {
7        int m;
8        for (int i=0; i<2; i++) {
9           System.out.println(id + " pede para retirar ");
10          m = b.remova();
11          System.out.println(id + " recebeu " + m);
12          yield();
13       }
14    }
15 }
```

E a família do produtores é dada pela classe **Produtor**:

```
1  class Produtor extends Thread {
2     private Buffer b; private String id;
3     public Produtor(String id, Buffer b) {
4        this.b = b; this.id = id ;
5     }
6     public void run() {
7        int m;
8        for (int i = 1; i<3; i++) {
9           m = 100 + i;
10          System.out.println(id + " quer depositar " + m);
11          b.insira(m);
12          yield();
13       }
14    }
15 }
```

O programa **Principal** inicia a computação alocando um *buffer* com capacidade para três inteiros e criando cinco linhas de consumidores e cinco de produtores, que são disparadas imediatamente.

```
1  public class Principal {
2    public static void main(String args[]) {
3        Buffer b = new Buffer(3);
4        Consumidor[] c = new Consumidor[5];
5        Produtor[]   p = new Produtor[5];
6        for (int i=1; i<5; i++) {
7           c[i] = new Consumidor("C" + i, b); c[i].start();
8           p[i] = new Produtor  ("P" + i, b); p[i].start();
9        }
10   }
11 }
```

E **Principal** produz a seguinte saída da Fig. 17.2, na qual ressalva-se que a ordem das ações impressas pode ser distinta em diferentes execuções, até mesmo no mesmo computador, porque a escalação de linhas foge ao controle do programa. Em alguns sistemas, uma linha monopoliza o controle até seu término. Em outros, o sistema implementa uma política de tempo compartilhado, ou seja, toda linha é periodicamente interrompida para que todas recebam, ciclicamente, uma fatia de tempo para execução.

Entretanto, quando se programa na rede, não se sabe em qual ambiente o programa com as linhas será executado. Assim, recomenda-se que toda linha chame **yield** ou **sleep** quando estiver executando um *loop* longo, assegurando-se que não estará monopolizando o processador do sistema. De qualquer forma, as ações de inserção e remoção atendem os requisitos do problema proposto.

```
C1 pede para retirar
P1 quer depositar 101
Inserido 101
Removido 101
C1 recebeu 101
P1 quer depositar 102
Inserido 102
C1 pede para retirar
Removido 102
C1 recebeu 102
C2 pede para retirar
P2 quer depositar 201
Inserido 201
Removido 201
C2 recebeu 201
C3 pede para retirar
P3 quer depositar 301
Inserido 301
Removido 301
C3 recebeu 301
P2 quer depositar 302
Inserido 302
C2 pede para retirar
Removido 302
C2 recebeu 302
P3 quer depositar 302
Inserido 302
C3 pede para retirar
Removido 302
C3 recebeu 302
C4 pede para retirar
P4 quer depositar 401
Inserido 401
Removido 401
C4 recebeu 401
P4 quer depositar 402
Inserido 402
C4 pede para retirar
Removido 402
C4 recebeu 402
```

Figura 17.2 Saída do Programa Principal

17.4 Encerramento de linhas

Pode-se determinar se uma linha está viva, isto é, se ainda não terminou, pela chamada ao seu método `isAlive()`. Uma linha termina somente quando seu `run` termina, seja normalmente ou por uma exceção que não foi capturada. Antigamente, era possível terminar linhas com o comando `stop()`, mas esse recurso foi depreciado e não deve mais ser usado.

Se uma linha `t1` deseja que uma outra `t2` encerre sua execução, `t1` pode enviar seu pedido a `t2` via a chamada `t2.interrupt()`, a qual gera a exceção `InterrupedException` na linha `t2`. Essa exceção somente é percebida por `t2` se ela estiver bloqueada por

alguma razão. Caso a linha `t2` esteja em execução no momento que a exceção foi levantada, o pedido de interrupção não será percebido, mas será anotado no estado de `t2`, podendo ser recuperado mais tarde via o comando `t2.interrupted()` ou `isInterrupted()`.

As operações da classe `Thread` que tratam da geração de exceções visando a interrupção de linhas são:

- `void interrupt() throws InterruptException`:
  linha corrente causa o levantamento da exceção indicada na linha do objeto receptor, o qual deve estar no estado dormente ou de espera.
- `static boolean interrupted()`:
  informa se a linha corrente recebeu algum pedido de interrupção ainda não inspecionado ou tratado, e faz o status de `interrupted` igual a `false`.
- `boolean isInterrupted()`:
  informa se a linha receptora da operação recebeu algum pedido de interrupção ainda não inspecionado ou tratado.

Considere a classe `T` abaixo, que após avisar que foi iniciada, vai dormir por três segundos, aguardando ser interrompida:

```
1  import java.io.*;
2  class T extends Thread {
3     public void run() {
4        System.out.println("T1. Inicia");
5        try {sleep(3000);}
6        catch(InterruptedException d) {
7             System.out.println("T2. Interrompida");
8        }
9        System.out.println("T4. Fim");
10    }
11 }
```

O programa **Interrompe**, apresentado a seguir, produz uma saída que mostra, passo-a-passo, o efeito causado na linha `t` pelo comando `t.interrupt` usado na linha 12, que causa o levantamento da exceção **InterruptedException**, que é capturada por `t` ainda adormecida devido ao longo **sleep** da linha 5 da classe `T`.

```
1  public class Interrompe {
2     static public void main(String[] args) {
3        PrintStream o = System.out;
4        T t = new T();
5        o.println("M1. Inicia");
6        t.start();
7        o.println("M2. Continua");
8        try {Thread.sleep(1000);}
9        catch(InterruptedException d) {
10           o.println("M3. Interrompida");}
11       o.println("M4. Continua");
12       t.interrupt();
13       o.println("M5. Fim");
14    }
15 }
```

A saída produzida é:

```
M1. Inicia
M2. Continua
T1. Inicia
M4. Continua
M5. Fim
T2. Interrompida
T4. Fim
```

Observe que o linha principal inicia a executa sem interrupção até a linha 8, quando entra em estado de dormência por 1000 ms. E somente nesse ponto, a linha `t` ganha o controle de execução e avança até a linha 5 para aguardar ser interrompida.

17.5 Grupos de linhas

Linhas podem ser agrupadas de forma a ter seus comportamentos controlados conforme certas regras, por exemplo, pode-se impor limites nas prioridades de execução das linhas de um grupo ou, com um único comando, gerar uma exceção de interrupção a todas de um mesmo grupo.

A classe `TreadGroup`, usada em certas construtoras da classe `Thread`, define as várias operações sobre o comportamento de linhas em grupos, os quais podem formar uma hierarquia conforme especificado na sua criação. Suas construtoras são as seguintes:

- `ThreadGroup(String s)`:
  constrói um novo grupo com o nome `s` e cujo pai é o grupo da linha em execução.
- `ThreadGroup(ThreadGroup pai, String s)`:
  constrói um novo grupo de nome `s` e filho do grupo `pai`.

As principais operações de `ThreadGroup` são as seguintes:

- `ThreadGroup getParent()`:
  informa o pai do grupo.
- `void destroy()`:
  destrói o grupo e seus subgrupos.
- `int getMaxPriority()`:
  obtém a prioridade máxima permitida às linhas do grupo.
- `void interrupt()`:
  gera uma exceção de interrupção em todas as linhas do grupo e de seus subgrupos.
- `boolean isDestroyed()`:
  testa se o grupo foi ou não destruído.
- `void list()`:
  imprime as informações a respeito do grupo.

- `boolean parentOf(ThreadGroup g)`:
  testa se o grupo é o mesmo que `g` ou um de seus ancestrais.
- `void setDaemon(boolean daemon)`:
  define o tipo *daemon* ou *user* para todas as linhas do grupo.
- `void setMaxPriority(int pri)`:
  define a prioridade máxima das linhas do grupo.

No momento de criação de uma linha pode-se alocá-la em grupos específicos, para os quais definem-se certas limitações ou operações, que são aplicáveis a todos os seus membros. Linhas de diferentes grupos podem ter limitações distintas, como restrição de modificar linhas de outros grupos. As construtoras de `Thread` relacionadas com vinculação de linhas em grupos são as seguintes:

- `Thread(ThreadGroup g, Runnable r, String s)`:
  constrói uma nova linha de nome interno `s`, fluxo de execução dado pelo método `run` de `r` e vinculada ao grupo `g`.
- `Thread(ThreadGroup g, Runnable r)`:
  constrói uma nova linha equivalente a `Thread(g,r,s)`, onde `s` é um nome gerado conforme o padrão `"Thread-"+ n`, onde `n` é um inteiro, iniciado com `0` e incrementado a cada nova linha alocada.
- `Thread(ThreadGroup g, String s)`:
  constrói uma nova linha equivalente a `Thread(g,null,s)`.
- `Thread(ThreadGroup g,Runnable r,String s,long t)`:
  constrói uma nova linha equivalente a `Thread(g,r,s)` e com uma pilha de tamanho `t`.
- `Thread(ThreadGroup g, Runnable r, String s, long t, boolean h)`:
  constrói uma nova linha equivalente a `Thread(g,r,s,t)`, se `h` igual a `true`. Se `h` for `false`, valores iniciais de variáveis locais não são herdados.

17.6 Exceções em linhas

As exceções não-tratadas de uma linha são a ela confinadas. No instante em que uma linha termina, suas exceções não-tratadas deixam de existir. Isso é uma consequência natural do fato de o método `run` de uma linha nunca retornar, pois linhas não têm ponto de retorno definido. Lembre-se que a operação `start` que dispara uma linha sempre retorna imediatamente, antes de a linha disparada iniciar efetivamente a executar o seu `run`. Portanto, não há meios de definir um `try-catch` que encapsule a execução de uma linha e trate suas exceções.

O bom estilo de programação recomenda que toda linha assegure que todas as exceções levadas por ela sejam devidamente capturadas e tratadas antes de decidir pelo seu encerramento.

17.7 Conclusão

Linhas permitem implementar sistemas que usam técnicas e recursos de Programação Concorrente com o objetivo de garantir acesso sincronizado e exclusivo no tempo a dados comuns e de tornar o provimento de serviços de forma mais eficiente do que seria se fosse feito de forma sequencial, uma tarefa por vez.

Este capítulo é apenas uma introdução à programação concorrente em Java, que deve ser suficiente para escrever os primeiros programas usando esses sofisticados recursos.

Recomenda-se a leitura da bibliografia citada para o bom domínio da matéria.

Exercícios

1. Qual é a utilidade de programação multilinhas?

Notas bibliográficas

Recomenda-se a leitura do livro de programação concorrente *Concurrent Programming in Java: Design Principles e Patterns* [14], do livro do Ken Arnold et alii [2], que tem um excelente capítulo sobre linhas e da página da Oracle para mais detalhes.

Capítulo 18

GUI: Componentes Básicos

`AWT` (*Abstract Windowing Toolkit*) é um pacote Java que disponibiliza uma série de classes relacionadas com a produção de interfaces gráficas usando os recursos da plataforma local. O `Swing` é um pacote gráfico, mais moderno que o `AWT`, cujos componentes são escritos totalmente em Java. Esses pacotes formam uma biblioteca de classes para implementação de interfaces gráficas e constituem a hierarquia da Fig. 18.1.

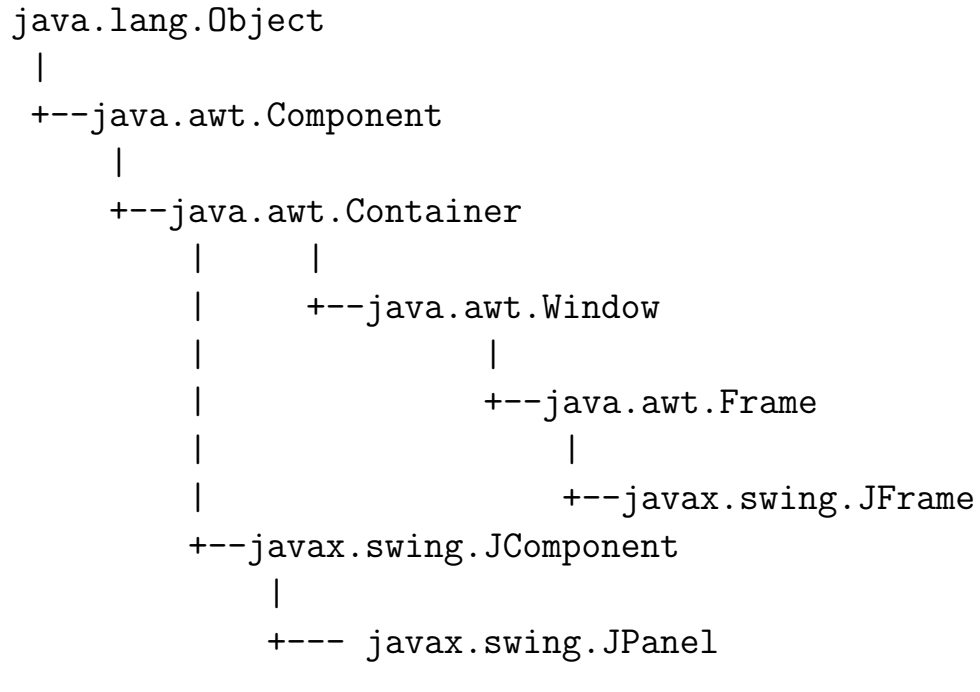


Figura 18.1 Hierarquia de `JFrame`

Em geral o comportamento de componentes `AWT` depende da plataforma na qual o programa é executado, ao passo que o comportamento dos componentes `Swing` são independentes de plataforma.

O elemento fundamental para implementação de uma interface é a janela, que é um objeto do programa com dados a ser exibidos na tela do computador. O conteúdo desse objeto descreve como a janela e seus componentes devem ser apresentados.

18.1 Classe `JFrame`

`JFrame` é a classe raiz de todas as janelas, as quais possuem decorações como bordas, barra de título e botões. Pode-se desenhar sobre um componente `JFrame` e também adicionar-lhe outros componentes gráficos, como botões, rótulos e caixas de texto.

Uma janela pode ser vista como um sistema de coordenadas, com seus valores, sempre positivos, dados em *pixel*, e cuja origem é o canto superior esquerdo da tela, como mostrado na Fig. 18.2.

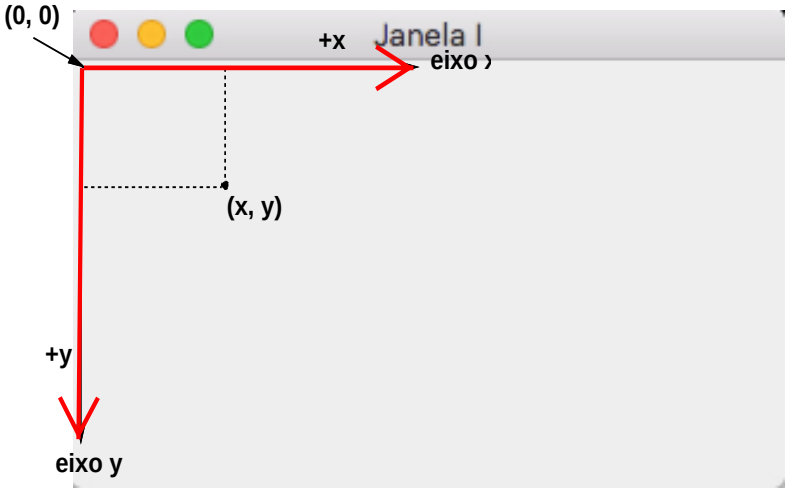


Figura 18.2 Sistema de Coordenadas da Tela

Para criar uma janela, deve-se seguir o seguinte protocolo:

- criar uma subclasse de `JFrame`, cuja construtora deve inicializar alguns atributos da janela, como título, tamanho, sua posição na tela, por meio das operações herdadas de `JFrame`;
- criar um objeto dessa subclasse, que tipicamente tem o efeito de criar a imagem da janela na memória;
- executar a operação `setVisible` para que a janela criada seja exibida na tela.

A classe `JFrame` oferece um grande número de operações para dar suporte à exibição de desenhos geométricos em janelas. Essas operações são aqui apresentadas ao longo deste capítulo. Algumas dessas operações são as seguintes:

- `JFrame()`:
  constrói janela inicialmente invisível.
- `JFrame(String t)`:
  constrói janela inicialmete invisível e com título `t`.
- `super(String t)`:
  chama construtora da classe ascendente com o título `t`.
- `setTitle(String t)`:
  define título da janela.
- `setSize(int largura, int altura)`:
  define dimensões da janela em pixels.
- `setSize(Dimension d)`:
  define dimensões da janela, a classe `Dimension` essencialmente encapsula largura e altura.
- `setVisible(boolean b)`:
  controla visibilidade da janela.
- `setLocation(int x, int y)` ou `setLocation(Point p)`:
  posiciona a janela na tela.
- `void setFont(Font f)`:
  define o fonte corrente da janela.

A janela da Fig. 18.3, que é uma janela decorada apenas por um título, é a criada e exibida pelo programa `UsaPrimeiraJanela`.

Figura 18.3 Primeira Janela

E o programa `UsaPrimeiraJanela` é o seguinte:

```
1  import javax.swing.*;
2  class Janela extends JFrame {
3     public Janela () {
4        setTitle("Janela I");
5        setSize(300,200);
6     }
7  }
8  public class UsaPrimeiraJanela {
9     public static void main(String [] args) {
10       JFrame janela = new Janela();
11       janela.setVisible(true);
12    }
13 }
```

Toda janela dispõe de três botões no canto superior direito ou esquerdo, que podem ser programados para reagir a cliques. Para isso, deve-se associar um objeto do tipo *ouvinte* a cada um desses botões. Na Seção 19, a construção de ouvintes está detalhada, e o exemplo a seguir apenas antecipa um de seus usos.

Nesse contexto, para criar a janela da Fig. 18.4, na qual um clique no botão **x** ou vermelho da barra superior comanda o seu fechamento, deve-se associar-lhe um ouvinte adequado.

Figura 18.4 Segunda Janela

```
1  import javax.swing.*; import java.awt.event.*;
2  class Janela extends JFrame {
3     public Janela (){
4        setTitle("Janela II");  setSize(300,200);
5        setVisible(true);
6     }
7  }
8  class Ouvinte extends WindowAdapter{
9     public void windowClosing (WindowEvent e){
10       System.exit(0);
11    }
12 }
13 public class UsaSegundaJanela {
14    public static void main(String [] args) {
15       JFrame janela = new Janela();
16       Ouvinte a = new Ouvinte();
17       janela.addWindowListener(a);
18    }
19 }
```

Um ouvinte de janela tem várias operações, dentre elas, a operação `windowClosing`, herdada de `WindowAdapter`, que deve ser redefinida para atribuir a semântica desejada ao ouvinte. Na implementação acima, o efeito do clique o botão `x` é a de apenas encerrar o programa via `System.exit(0)`. Para isso, define-se na linha 8 a classe `Ouvinte` como uma subclasse de `WindowAdapter` e redefine-se o método `windowClosing` apropriadamente e, posteriormente, nas linhas 16 e 17, associa-se um objeto da classe `Ouvinte` à janela usando a operação `AddWindowListener` de `JFrame`.

Quando um evento associado ao ouvinte, no presente caso quando um clique na janela ocorrer, a operação do ouvinte relacionada com o evento será automaticamente acionada. Veja os detalhes no programa `UsaSegundaJanela` apresentado a seguir.

Como o ouvinte criado nesse exemplo é um objeto usado apenas uma vez no programa, pode-se simplificar sua codificação eliminando as instruções de criar uma classe, criar um objeto dessa classe e associá-lo à janela, da seguinte forma:

```
1  public class UsaSegundaJanela {
2     public static void main(String [] args) {
3        JFrame janela = new Janela();
4        janela.addWindowListener(new WindowAdapter{
5           public void windowClosing (WindowEvent e){
6              System.exit(0);
7           }
8        });
9     }
10 }
```

onde o objeto ouvinte é criado a partir de uma classe anônima, conforme descrito na Seção 10.4

Outra solução, ainda mais simples, é usar o tratamento *default* para fechamento de janela:

```
1 import javax.swing.JFrame;
2 public class UsaSegundaJanela {
3   public static void main( String args[]) {
4     JFrame janela = new Janela();
5     janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6   }
7 }
```

onde `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` é uma operação da classe `JFrame` que define a ação *default* para clique no botão vermelho, que, no caso, simplesmente fecha a janela.

Esse programa simplesmente cria um objeto da classe `Janela`, e associa-lhe um ouvinte de seus eventos. Essa implementação de programa principal será usado como modelo padrão nos capítulos deste livro que tratam de componentes GUI.

18.1.1 Operação `paint`

Há operações de `JFrame` que não devem ser usadas diretamente no programa, pois têm funções específicas para tratar certos eventos. Por exemplo, a função `paint` é chamada automaticamente em resposta a eventos como clique de *mouse* ou abertura de uma janela. Quando isso ocorre, é passado a esse método um objeto do tipo `Graphics`, no qual tudo que lhe for escrito aparecerá na janela que gerou o evento. Se chamadas à `paint` forem necessárias, o protocolo exige que `repaint` seja chamada em seu lugar.

As operações `paint`, que é herdada da classe `Container`, e `repaint`, definida diretamente em `JFrame`, operam da seguinte forma:

- `paint(Graphics g)`:
  - desenha na janela os objetos gráficos definidos no objeto passado ao parâmetro `g`;

  - é chamada automaticamente, por exemplo, em resposta a um *evento*, como abertura de janela;
  - deve ser redefinida pelo programa do usuário de forma a construir o conteúdo do objeto `g`;
  - não deve ser chamada diretamente.

- `repaint()`:
  - nunca é chamada automaticamente, deve ser chamada pelo usuário, quando necessário;
  - limpa o fundo do componente gráfico corrente de qualquer desenho anterior e chama o método `paint` para redesenhar as informações gráficas sobre tal componente.

18.2 Classe `Graphics`

`Graphics` é uma classe abstrata que especifica as operações usadas para desenhar formas sobre componentes gráficos. Cada ambiente Java implementa uma classe derivada de `Graphics` com todos os recursos de desenho. Internamente em cada programa são criados objetos dessa classe para controlar como os desenhos são feitos. Esses são os objetos que são passados automaticamente ao `paint` nos momentos apropriados.

`Graphics` disponibilizam mais de 50 operações que visam desenhar formas geométricas, manipular estilos, tamanhos, tipos de fontes e cores, escrever textos, etc. Todos os desenhos e textos escritos na janela usa a cor, fontes e estilo correntes ou atuais.

Dentre as operações destinadas a desenhos geométricos, destacam-se as seguintes:

- `void drawLine(int x1, int y1, int x2, int y2)`: desenha uma linha ligando os pontos `(x1,y1)` e `(x2,y2)`.

- `void drawRect(int x,int y,int largura,int altura)`: desenha um retângulo com a largura e a altura especificadas.
- `void fillRect(int x,int y,int largura,int altura)`: desenha um retângulo sólido com a largura e altura especificadas, com canto superior esquerdo no ponto `(x,y)`.
- `void clearRect(int x,int y,int width,int height)`: desenha um retângulo transparente, ou seja, com a cor da borda e a cor de fundo igual a cor do componente sobre o qual será desenhado. Usado para separar componentes.
- `drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`: Desenha um retângulo com cantos arredondados.
- `void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`: desenha um retângulo sólido, preenchido com a cor atual e com cantos arredondados.
- `void draw3DRect(int x, int y, int width, int height, boolean b)`: desenha um retângulo tridimensional em alto relevo quando `b` for `true` e em baixo relevo quando `b` for `false`.
- `void fill3DRect(int x, int y, int width, int height, boolean b)`: desenha um retângulo tridimensional preenchido com a cor atual em alto ou em baixo relevo de acordo com o valor do parâmetro `b`.
- `void drawOval(int x,int y,int width,int height)`: desenha uma oval, como na Fig. 18.5, cujos cantos tocam a parte central das bordas de um retângulo de canto superior esquerdo em `(x,y)`, tendo largura e altura especificadas por `width` e `height`, respectivamente.

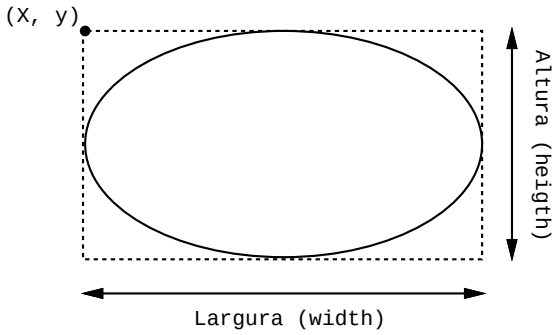


Figura 18.5 Oval

- `void fillOval(int x,int y,int width,int height)`: Desenha uma oval com o fundo preenchido na cor atual.

A interface `Graphics` é usada na definição do método `paint` da classe `JFrame`, que é chamado automaticamente quando um componente dessa classe é criado. O programador não pode chamar `paint`, mas deve defini-lo adequadamente.

Para exemplificar o uso de `Graphics`, o programa `UsaVitral` a seguir constrói a janela da Fig. 18.6, intitulada `Vitral de Formas`, que aceita clique no botão vermelho, para encerrar a aplicação, e que desenha cinco objetos, que são, pela ordem: uma linha horizontal, um retângulo transparente, um retângulo cheio, um retângulo de cantos arredondados e um outro retângulo transparente.

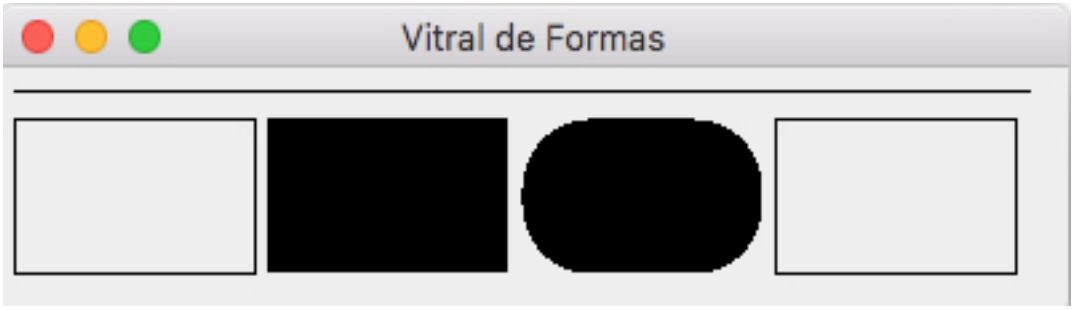


Figura 18.6 Vitral de Formas

A classe `Vitral` apresentada a seguir define uma janela como a do exemplo `UsaSegundaJanela` apresentado na Seção 18.1, sendo

a novidade a redefinição do método **paint** na linha 10, que é automaticamente chamado assim que a janela for criada, para preencher o objeto **g** com as figuras geométricas desejadas.

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class Vitral extends JFrame {
4      public Vitral() {
5          super ("Vitral de Formas");
6          setSize (400, 110);
7          setVisible(true);
8      }
9
10     public void paint (Graphics g) {
11         g.drawLine (5, 30, 380, 30);
12         g.drawRect (5, 40, 90, 55);
13         g.fillRect (100, 40, 90, 55);
14         g.fillRoundRect (195, 40, 90, 55, 50, 50);
15         g.draw3DRect (290, 40, 90, 55, true);
16     }
17 }
```

O programa principal segue o padrão já apresentado de criar a janela e associar-lhe um ouvinte para o seu botão de fechamento de janela:

```
1  import javax.swing.JFrame;
2  public class UsaVitral {
3    public static void main( String args[]) {
4      JFrame janela = new Vitral();
5      janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6    }
7  }
```

Para escrita de texto na janela, controle das cores e dos fontes usados, **Graphics** disponibiliza as seguintes operações, que podem ser usadas pelo método **paint**:

- `void drawString(String texto, int x, int y)`:
  desenha o `texto` na posição `(x,y)` da janela com a fonte e color correntes.
- `Color getColor()`:
  devolve um objeto Color que representa a cor de textos usada nas operações.
- `void setColor(Color c)`:
  configura a cor atual para desenho de textos e linhas.
- `Font getFont()`:
  retorna uma referência de objeto `Font` representando a fonte atual.
- `void setFont(Font f)`:
  configura a fonte atual com a fonte, o estilo e o tamanho especificados pela referência ao objeto `Font f`.

18.3 Classe `Font`

A plataforma Java apresenta dois tipos de fontes: o físico e o lógico. O físico consiste em uma biblioteca que descreve a representação básica de cada caractere dependente de ambiente de execução, enquanto o fonte lógico apresenta família de fontes independentes de plataformas, no qual caracteres têm fonte, estilo e tamanho. Exemplos de fontes são `Serif`, `SansSerif`, `Monospaced`, `Dialog` e `DialogInputTimes`, `Helvetica`, `Arial`, `Serif`, `SansSerif`, `Dialog` e `Monospaced`.

O estilo do fonte pode ser `PLAIN`, `ITALIC` ou `BOLD` ou uma combinação desses estilos.

E o tamanho é um inteiro denotando o número de pontos do caractere, sendo 1 ponto igual a 1/72 da polegada.

Os objetos que descrevem fontes são da classe `Font`, a qual

possui as seguintes operações[1]:

- `static int PLAIN`:
  uma constante representando um estilo de fonte simples.
- `static int BOLD`:
  uma constante representando um estilo de fonte em negrito.
- `static int ITALIC`:
  uma constante representando um estilo de fonte em itálico.
- `Font(String fonte, int estilo, int tamanho)`:
  cria um objeto `Font` com a fonte, o estilo e o tamanho especificados na construtora.
- `int getSize()`:
  retorna o tamanho da fonte.
- `String FontName()`:
  retorna o nome da fonte, por exemplo, `"Helvetica Bold"`.
- `int getStyle()`:
  retorna o estilo da fonte, por exemplo, `"Bold"`.
- `String getName()`:
  retorna o nome lógico da fonte, por exemplo, `"SansSerif"`.
- `String getFamily()`:
  retorna o nome da família da fonte, e.g.,`"Helvetica"`.
- `boolean isPlain()`:
  testa uma fonte quanto ao estilo de fonte simples. Retorna `true` se o estilo da fonte é simples (PLAIN).
- `boolean isBold`:
  testa uma fonte quanto ao estilo de fonte em negrito. retorna `true` se a fonte estiver em negrito.
- `boolean isItalic()`:
  testa uma fonte quanto ao estilo de fonte em itálico. Retorna `true` se a fonte estiver em itálico.

[1]Constantes são consideradas operações sem operandos

O programa **UsaDesenhaFrase** apresentado a seguir mostra como redefinir o método **paint** de **DesenhaFrase** para escrever duas frases nessa janela. Para isso, são usadas as operações **setFont**, para definir o fonte corrente da janela e **drawString**, que escreve uma frase conforme o fonte corrente.

A primeira frase, iniciando nas coordenadas `(20,50)`, tem fonte *Monospaced*, estilo itálico e tamanho 20 pontos, e a outra, nas coordenadas `(20,70)`, em uma fonte *Helvetica*, em negrito e com tamanho 40 pontos, conforme mostra na Fig. 18.7.

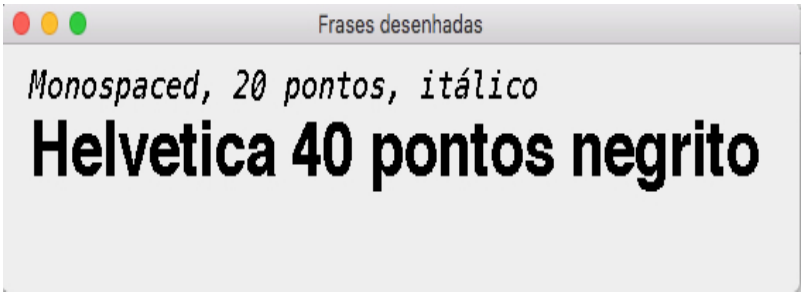


Figura 18.7 Escrita Gráfica

```
1  import java.awt.*; import javax.swing.*;
2  import java.awt.event.*;
3  public class DesenhaFrase extends JFrame {
4     public DesenhaFrase() {
5       super ("Frases desenhadas");
6       setSize (420,150); setVisible(true);
7     }
8     public void paint (Graphics g) {
9       g.setFont(new Font ("Monospaced", Font.ITALIC,20));
10      g.drawString("Monospaced, 20 pontos italico",20,50);
11      g.setFont(new Font("Helvetica",Font.BOLD,40));
12      g.drawString("Helvetica 40 point negrito",20,90);
13    }
14 }
```

O programa principal é o seguinte:

```
1 public class UsaDesenhaFrase {
2   public static void main (String args[]) {
3     JFrame j = new DesenhaFrase();
4     j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5   }
6 }
```

18.4 Classe `Color`

As operações de desenho de figuras geométricas ou de escrita de textos da classe **`Graphics`** usam um conjunto de parâmetros de apresentação e de estilo que podem ser modificados a qualquer momento no programa. Um desses parâmetros é a cor usada nas figuras produzidas, que pode ser definida via as seguintes operações da classe **`Color`**:

- **`Color (int r, int g, int b)`**:
  cria uma cor com base na quantidade de vermelho, verde e azul, expressos como inteiros entre 0 e 255.
- **`Color (float r, float g, float b)`**:
  cria uma cor com base na quantidade de vermelho, verde e azul, expressos como valores de ponto flutuante entre 0.0 e 1.0.
- *nome da cor*, que pode ser uma das seguintes constantes:

Constante Color	Cor	Valor RGB
`final static Color orange`	laranja	255,200,0
`final static Color cyan`	ciano	0,255,255
`final static Color magenta`	magenta	255,0,255
`final static Color yellow`	amarelo	255,255,0
`final static Color black`	preto	0,0,0
`final static Color white`	branco	255,255,255
`final static Color gray`	cinza	128,128,128
`final static Color lightGray`	cinza-claro	192,192,192
`final static Color darkGray`	cinza-escuro	64,64,64
`final static Color red`	vermelho	255,0,0
`final static Color green`	verde	0,255,0
`final static Color blue`	azul	0,0,255

- `int getRed()`:
  retorna um valor entre 0 e 255 representando o conteúdo de vermelho.
- `int getGreen()`:
  retorna um valor entre 0 e 255 representando o conteúdo de verde.
- `int getBlue()`:
  retorna um valor entre 0 e 255 representando o conteúdo de azul.

O programa abaixo cria a janela da Fig. 18.8, que exibe um retângulo vermelho e outro verde.

```
1  import java.awt.*;  import javax.swing.*;
2  import java.awt.event.*;
3  public class RectColorido extends JFrame {
4      public RectColorido() {
5        super ("Retângulos coloridos");
6        setSize (150, 80); setVisible(true);
7      }
8    public void paint (Graphics g) {
9      g.setColor(new Color (255,0,0)); // vermelho
10     g.fillRect(25, 25, 100, 20);
11     g.setColor(new Color(0.0f,1.0f,0.0f ));// verde
12     g.fillRect(25, 50, 100, 20);
13   }
14 }
15 public class UsaRectColorido {
16   public static void main (String args[]) {
17     RectColorido j = new RectColorido();
18     j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19   }
20 }
```

Figura 18.8 Uso de Cores

18.5 Classes `JLabel`, `JButton` e `JTextField`

Objetos da classe **`Graphics`** são muito úteis para desenhar formas geométricas e escrever textos coloridos em janelas da interface, entretanto não abordam formas de o usuário interagir com componentes da interface gráfica. Esses componentes são objetos com os quais o usuário interage via mouse, teclado, voz, etc.

Essa capacidade de interação do **`Swing`** é oferecida pela classe **`Component`** e suas subclasses. E os componentes mais básicos de uma interface gráfica são rótulos, botões e áreas de texto, como os mostrados na Fig. 18.9.

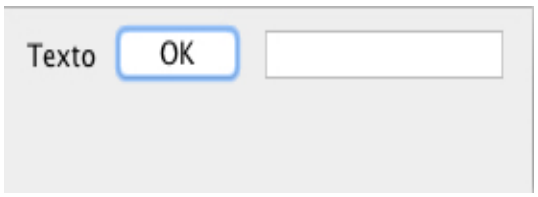


Figura 18.9 Componentes Básicos

Outros componentes muito comuns são caixas de seleção, lista e painéis. Para modelar cada um desses objetos há as classes **`JLabel`**, **`JButton`**, **`JTextField`**, **`JComboBox`**, **`JList`** e **`JPanel`**, que estão detalhadas a seguir.

A classe **`JLabel`** permite criar rótulos, que são áreas em que podem ser exibidos textos não-editáveis ou ícones. Tipicamente, rótulos são utilizados para exibir um pequeno texto que não pode

ser alterado e são usados para nomear botões e caixas de texto. E os métodos de `JLabel` são os seguintes:

- `JLabel()`:
  cria um rótulo sem identificação.
- `JLabel(String t)`:
  cria rótulo com identificação `t`.
- `void setText(String t)`:
  define `t` com a identificação do rótulo.
- `String getText()`:
  devolve a identificação do rótulo.

A classe `JButton` permite criar um botão, que é uma área na janela que aciona um evento quando recebe um clique de *mouse*. Os métodos da classe `JButton` são:

- `JButton()`:
  cria botão sem identificação.
- `JButton(Icon icon)`:
  cria botão com o ícone `icon`.
- `JButton(String texto)`:
  cria botão identificado com o texto dado.

A classe `JTextField` permite criar caixas de textos, que são um recurso utilizado nas interfaces gráficas para receber texto do usuário ou para transmitir-lhe um texto. A digitação da tecla *Enter* gera um evento, que dá a oportunidade de se dar a destinação desejada aos dados digitados na caixa de texto.

O tratamento dos eventos gerados por botões e caixas de texto é discutido na Seção 19.

As principais construtoras de `JTextField` são:

- `JTextField()`
- `JTextField(int colunas)`
- `JTextField(String texto)`

18.6 Classes JComboBox, JList, JPanel

Um objeto `JComboBox` é uma caixa de combinação que consiste em uma lista *drop-down* de itens a partir da qual o usuário pode fazer uma seleção clicando em um item da lista ou digitando um texto na caixa, se permitido.

Um objeto `JList` é uma área em que uma lista de itens é exibida e a partir da qual o usuário pode fazer uma seleção clicando uma vez em qualquer elemento na lista. Um clique duplo em um elemento na lista gera um evento de ação. Listas também permitem selecionar múltiplos elementos.

Um objeto `JPanel` é um painel, que consite em contêiner no qual componentes podem ser colocados.

O tratamento de eventos relativos a esses componentes está detalhado na Seção 19.3.1.

18.7 Anexação de componentes

Para utilizar efetivamente os componentes GUI, as hierarquias de classes e interfaces dos pacotes `javax.swing` e `java.awt` devem ser compreendidas. Em particular, deve-se conhecer especialmente as classes `JComponent` e `Container`, que definem os recursos comuns à maioria dos componentes Swing.

A hierarquia de `JComponent` e de `Container` estão esquematizadas nas figuras Fig. 18.10 e Fig. 18.11. E essas classes reúnem uma grande coleção de componentes gráficos e de mecanismos de colocação desses componentes nas janelas.

A classe `Container` define objetos onde componentes são anexados e `JComponent` declara muitos atributos e operações comuns a componentes da família `awt` e `swing`.

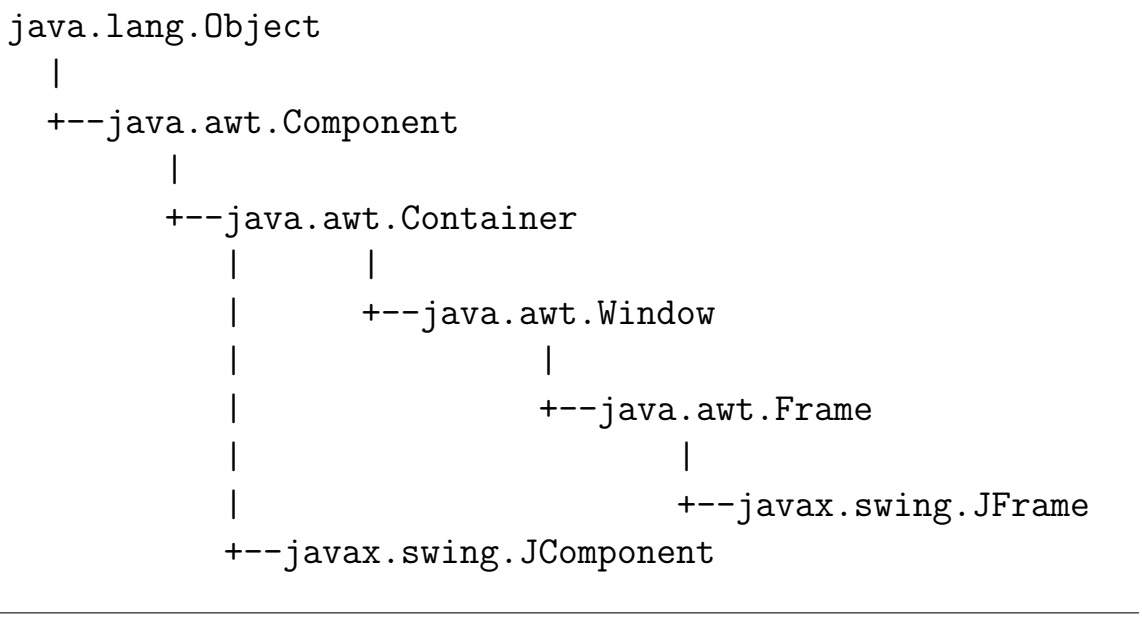


Figura 18.10 Hierarquia de `Component`

18.7.1 Classe `JComponent`

Fig. 18.11 mostra a hierarquia de `JComponent`, onde estão destacadas as presenças de componentes gráficos muito importantes como `JLabel`, `JButton` e `JtextField`, já estudados, e também `JPanel`, `JComboBox`, `JList` e outros que são abordados a partir da Seção 19.3.1.

18.7.2 Classe `Container`

Um objeto do tipo `Container` é uma coleção ou um contêiner de componentes associados a determinados componentes gráficos.

Certos componentes gráficos possuem uma área denominada camada ou painel de conteúdo, que é um objeto do tipo `Container`, e onde outros componentes gráficos podem ser posicionados. Esses painéis possuem um leiaute interno de posicionamento dos componentes, e que pode ser redefinido se necessário.

Uma janela `JFrame` é uma estrutura com várias camadas ou vi-

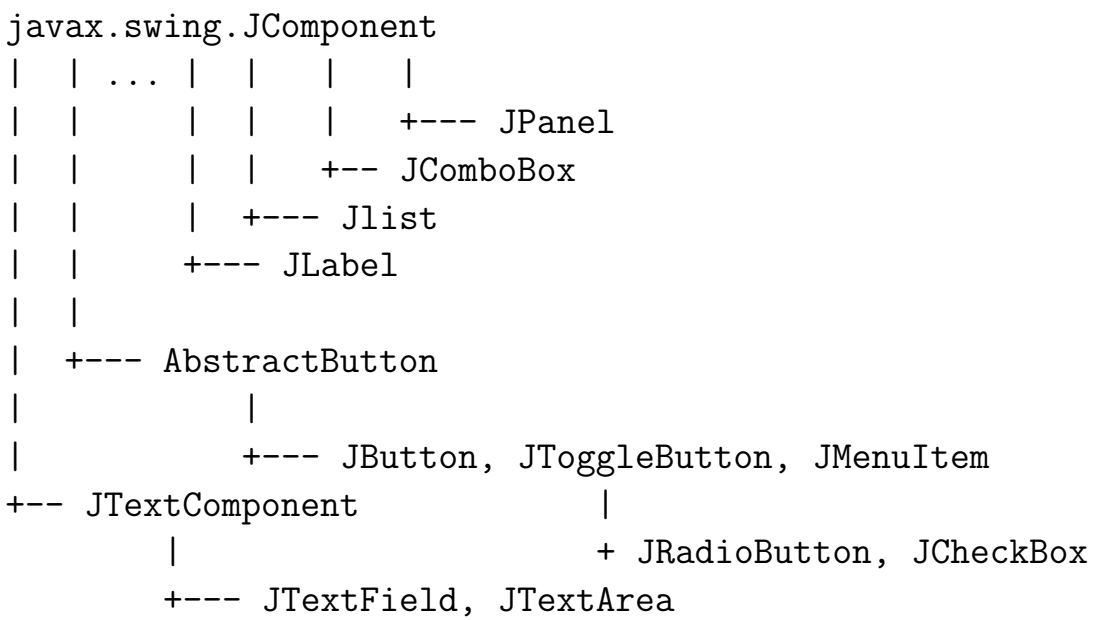


Figura 18.11 Hierarquia de `JComponent`

draças, que atendem a variados propósitos na configuração de uma interface, e que podem ser recuperadas pelas seguintes operações de `JFrame`:

- `Component getGlassPane()`:
  retorna o objeto camada de vidro da janela.
- `Container getContentPane()`:
  retorna o objeto camada de conteúdo da janela.
- `JLayeredPane getLayeredPane()`:
  retorna o objeto camada *layered* da janela.
- `JRootPane getRootPane()`:
  retorna o objeto camada raiz da janela.

A operação `getContentPane` de `JFrame` é o ponto de partida para obtenção da camada de conteúdo onde componentes podem ser anexados em uma janela por meio das operações disponibilizadas pela classe `Container`, dentre as quais destacam-se os seguintes métodos:

- `setLayout(LayoutManager leiaute)`:
  define o leiaute da camada de conteúdo para posicionamento de componentes, e.g., `FlowLayout` ou `BorderLayout`.
- `add(Component c)`:
  adiciona no fim da camada de conteúdo o componente `c`.
- `add(Component c, int posicao)`:
  adiciona à camada de conteúdo o componente `c` na posição indicada do contêiner.
- `setFont(Font f)`:
  define o fonte da camada de conteúdo.
- `setBounds(int x, int y, int largura, int altura)`:
  posiciona e define tamanho de um componente.

Há duas formas de posicionar componentes na camada de conteúdo de uma janela: posicionamento absoluto e uso de gerenciador de leiautes. No posicionamento absoluto, o programador tem total controle do processo, que é obtido configurando o gerenciador de leiaute do contêiner como `null`, i.e., executando um `setlayout(null)` para o contêiner da camada de conteúdo. A partir daí, a posição absoluta de cada componente em relação ao canto superior esquerdo deve ser informado, sendo também necessário que se especifique o seu tamanho.

O seguinte trecho de código ilustra as operações para colocar `meuComponente` na posição `(x,y)` da janela `f`, devendo o componente ocupar uma área de tamanho `largura` X `altura`:

```
JFrame f = new MinhaFrame();
Component meuComponente = new MeuComponente();
Container c = f.getContentPane();
c.setLayout(null);
meuComponente.setBounds(x,y,largura,altura);
c.add(meuComponente);
```

18.7.3 Gerenciadores de leiaute

Gerenciadores de leiaute se prestam a organizar componentes GUI em um contêiner. Ele retira do programador a tarefa de posicionar os itens em um contêiner, tornando o posicionamento automático. As principais classes para gerenciamento automático do leiaute são `FlowLayout`, `BorderLayout` e `GridLayout`.

A classe `FlowLayout` define que componentes sejam normalmente posicionados sequencialmente da esquerda para a direita e de cima para baixo na ordem em que forem adicionados. Com esse tipo de leiaute, também é possível especificar a ordem dos componentes utilizando o método `add` de `Container`, que aceita um `Component` e o índice de sua posição como argumentos.

O programa abaixo ilustra a adição de cinco botões em uma pequena janela, cujo modelo de leiaute foi definido na linha 7 da classe `Flow`, pela passagem de um objeto do tipo `FlowLayout` para o método `setLayout` para configurar o painel de controle.

```
1  import java.awt.*;
2  import javax.swing.*;
3  public class Flow extends JFrame {
4     Flow() {
5        this.setSize(190,200);
6        Container c = this.getContentPane();
7        c.setLayout(new FlowLayout());
8        c.add(new JButton("primeiro"));
9        c.add(new JButton("segundo"));
10       c.add(new JButton("terceiro"));
11       c.add(new JButton("quarto"));
12       c.add(new JButton("quinto"));
13       this.setVisible(true);
14    }
15 }
```

O programa `UsaFow` abaixo produz a saída da Fig. 18.12. A adição de componentes ocorre da esquerda para a direita e de cima para baixo, e que mudanças de linha ocorrem quando não houver, na linha corrente, espaço suficiente para o componente que está sendo adicionado. Os componentes são sempre centralizados.

```
1 import javax.swing.*;
2 public class UsaFlow {
3    public static void main (String args[]) {
4       JFrame j = new Flow();
5       j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6    }
7 }
```

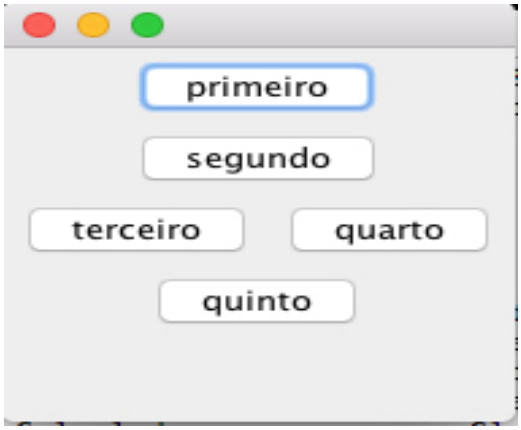


Figura 18.12 `FlowLayout`

O gerenciador de leiaute `BorderLayout` organiza os componentes em cinco áreas da camada de conteúdo, definidas, conforme a Fig. 18.13, por `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST` e `BorderLayout.CENTER`.

No momento da adição de cada componente na camada, deve-se informar em qual dessas áreas ele deve ser inserido. Cada componente ocupa um espaço conforme seu tamanho, deslocando sua área receptora se necessário.

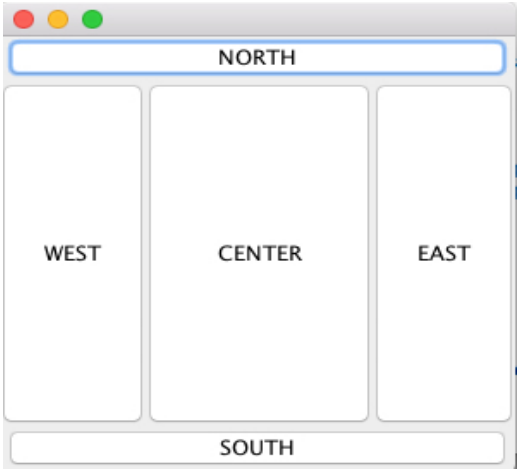


Figura 18.13 `BorderLayout`

O programa a seguir aloca cinco botões na janela, com o nome da região definida pelo **`BorderLayout`**, produzindo a Fig. 18.13.

```
1  import java.awt.*; import javax.swing.*;
2  import java.awt.event.*;
3  class Border extends JFrame {
4     public Border() {
5        Container c = this.getContentPane();
6        c.setLayout(new BorderLayout());
7        c.add(new JButton("NORTH"), BorderLayout.NORTH);
8        c.add(new JButton("SOUTH"), BorderLayout.SOUTH);
9        c.add(new JButton(" EAST "),BorderLayout.EAST);
10       c.add(new JButton(" WEST "),BorderLayout.WEST);
11       c.add(new JButton("CENTER"),BorderLayout.CENTER);
12       this.setSize(300,300); this.setVisible(true);
13    }
14 }
15 public class UsaBorder {
16    public static void main (String args[]) {
17       JFrame j = new Border();
18       j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19    }
20 }
```

Esse é o gerenciador padrão para os painéis de **JFrame**, i.e., se o gerenciador não for especificado, **BorderLayout** é usado.

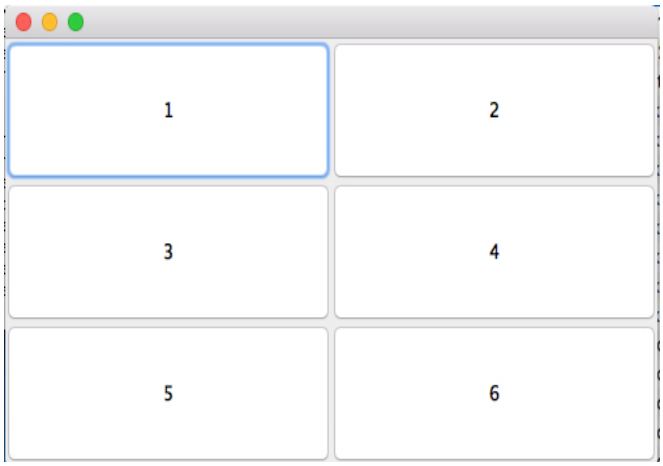


Figura 18.14 GridLayout

O gerenciador de leiaute **GridLayout** divide uma camada de conteúdo em **m** linhas e **n** colunas como em uma matriz **m** X **n**, conforme exemplificado na Fig. 18.14, que foi gerada pela construtora da classe **Grid**:

```
1  import java.awt.*;
2  import javax.swing.*;
3  import java.awt.event.*;
4  public class Grid extends JFrame {
5     public Grid() {
6         Container c = getContentPane();
7         c.setLayout(new GridLayout(3,2));
8         c.add(new Button("1"));  c.add(new Button("2"));
9         c.add(new Button("3"));  c.add(new Button("4"));
10        c.add(new Button("5"));  c.add(new Button("6"));
11        setSize(500,300);
12        setVisible(true);
13    }
14 }
```

A principal diferença entre **FlowLayout** e **GridLayout** é que neste gerenciador os componentes ocupam área de mesmo tamanho

na camada de conteúdo, i.e., as entradas na matriz são de tamanho fixo. Tal como no `FlowLayout`, componentes são colocados na camada de modo a preencher cada elemento da matriz da esquerda para a direita, de cima para baixo.

E a classe `UsaGrid` abaixo é o programa principal que cria a janela `Grid` e produz a saída da Fig. 18.14.

```
1 public class UsaGrid {
2    public static void main (String args[]) {
3       JFrame j = new Grid();
4       j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5    }
6 }
```

18.8 Conclusão

Os recursos para programação de interfaces gráficas em Java é extremamente rico com muitas dezenas de classes e centenas de métodos. Este capítulo apenas aborda importantes e básicos pontos dessa complexa biblioteca, que são fundamentais para qualquer iniciante nessa área. Para aprofundar o assunto, recomenda-se a leitura dos capítulos 19, 21 e 22 deste livro e da bibliografia arrolada, em especial os livros dos Deitels.

Exercícios

1. Escreva um aplicativo para iterativamente pedir ao usuário para digitar um valor em uma caixa de texto e então imprimir a raiz quadrada desse valor em um rótulo.

Notas bibliográficas

Um dos melhores texto para estudar e aprofundar programação de interfaces gráfica é o livro *Java - Como programar* dos Deitels[10]. Os capítulos 11, 12 e 22 desse livro apresentam muitos exemplos ilustrativos de uso dos pacotes `Swing` e `AWT`. Vale a pena ler.

Capítulo 19

GUI: Modelo de Eventos

Interfaces gráficas são um mecanismo que permite ao usuário interagir com programas que exibem um conjunto textos e objetos gráficos do tipo `JComponent` e reagem quando algum evento é gerado devido a ações do usuário.

As ações típicas do usuário no processo de interagir com a interface gráfica são: movimentar o *mouse*, pressionar o botão do *mouse*, pressionar da teclar *Enter* após a digitação de um campo de texto, provocar a abertura de uma janela e selecionar um item de menu. Essas ações geram eventos que devem ser devidamente processados pelo programa. Cada componente de uma interface pode ter a ele associado um conjunto tratadores de eventos, cada um com uma semântica própria.

Os tratadores de eventos são denominados *ouvintes*, e cada objeto de interface gráfica, originador de eventos, administra uma lista de ouvintes, de forma que, quando um evento ocorrer com esse objeto, sua lista de ouvintes é pesquisada para identificar os responsáveis para o seu tratamento.

Existem em Java 11 tipos de ouvintes, sendo cada tipo definido por uma interface Java, que especifica os métodos de tratamento de eventos associados a ser implementados.

A preparação de uma interface gráfica para o tratamento de eventos envolve os seguintes passos:

- criar as classes dos ouvintes desejados, implementando os tratadores (*handler*) de eventos, que são métodos especiais especificados nas interfaces Java associadas;
- criar os objetos ouvintes correspondentes;
- registrar esses ouvintes nos objetos originadores de eventos.

Um mesmo componente por ter mais de um tipo de ouvinte associado, pois mais de um tipo de evento pode ocorrer com o componente e um mesmo evento pode ser tratado por mais de um ouvinte.

19.1 Eventos e ouvintes

Um objeto ouvinte é instância de uma classe que implementa uma interface Java de nome `EventListener`, onde `Event` identifica o evento relacionado, i.e., `Action`, `Component`, `Container`, `Focus`, `Item`, `ListSelection`, `Mouse`, `MouseMotion`, `MouseWheel`, `Key` ou `Window`.

Os métodos especificados por essas interfaces devem implementar as ações para atendimento de cada evento. De forma que, quando um evento ocorre, o objeto que o originou prepara um descritor do evento, percorre sua lista de ouvintes para identificar os que são vinculados ao evento ocorrido, e, para cada um, chama o seu método associado ao evento, passando-lhe o objeto descritor do evento.

Por exemplo, os objetos ouvintes de eventos do tipo `ActionEvent`, que podem ser gerados por cliques em botões, devem ser de classes que implementam a interface `ActionListener`:

```
interface ActionListener {
   void actionPerformed (ActionEvent e);
}
```

onde o parâmetro `e` é o descritor do evento construído pelo objeto originador que ativa o método `actionPerformed` em atendimento à ocorrência do evento. Os métodos da classe `ActionEvent` que permitem inspecionar a descrição do evento são os seguintes:

- `getSource()`:
  retorna a referência do objeto originador do evento.
- `String getActionCommand()`:
  retorna a cadeia de caractere que identifica o comando associado com a ação, por exemplo, o rótulo de um botão.

Com frequência o método de tratamento de eventos implementado por `actionPerformed` deve produzir algum efeito nos dados relativos ao componente que gerou o evento e, por isso, precisa de ter acesso a esses dados, que pode ser feito transmitindo-se suas referências via o descritor de evento que é passado ao método ou definindo o ouvinte a partir de uma classe interna ao componente.

Todo objeto do tipo `JComponent`, originador de eventos, tem uma referência a uma lista de objetos ouvintes, que são os tratadores de eventos a ele associado. A inserção de um ouvinte `z` na lista de um componente `x` é realizada por `x.addEventListener(z)`, onde a palavra `Event` deve ser substituída pelo nome que identifica o evento relacionado, i.e., `Action`, `Component`, `Container`, `Focus`, `Item`, `ListSelection`, `Mouse`, `MouseMotion`, `Window`, `MouseWheel` ou `Key`.

Eventos do tipo `Action` podem ser gerados por objetos, como um `JButton` que recebe clique, um `List` ou um `MenuItem` que têm um item selecionado, ou um `JTextField` que recebe a digitação de um *Enter*.

Para cada caso, um objeto ouvinte do tipo `ActionListener` deve ser implementado e associado ao objeto originador do evento, conforme resumido na Fig. 19.1.

Interface	Métodos	Parâmetro/ Acessadores	Origem
ActionListener	actionPerfomed	ActionEvent getSource getActionCommand	Button List MenuItem TextField

Figura 19.1 Ouvinte de Eventos `Action`

Para exemplificar o uso desses recursos, considere o programa **`UsaEntradaDeTexto`** que exibe uma caixa de texto, na qual o usuário digitou **`Carolina`**, conforme mostrado na Fig. 19.2.

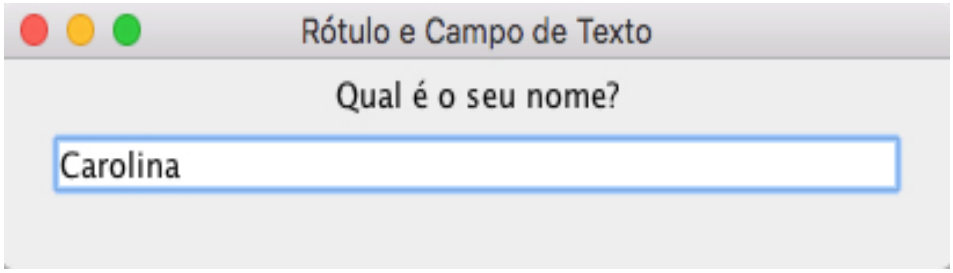


Figura 19.2 Tela do `UsaEntradaDTexto`

E, imediatamente, o programa responde com a janela da Fig. 19.3.

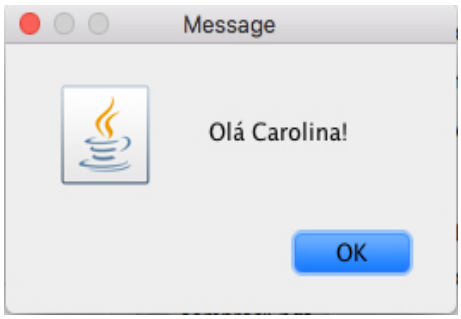


Figura 19.3 Tela do `UsaEntradaDTexto`

O programa **`UsaEntradaDeTexto`** cria a janela declarada com o tipo **`EntradaDeTexto`** na qual exibem-se os componentes **`JLabel`** e **`JTexField`** e implementa um tratador de eventos, que é associ-

ado, na linha 12 do código abaixo, ao objeto `texto` via o comando `addActionListener(new Ouvinte())`:

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  public class EntradaDeTexto extends JFrame {
4     private class Ouvinte implements ActionListener {...}
5     private JTextField texto = new JTextField (30);
6     private JLabel label = new JLabel("Qual é o seu nome?");
7     EntradaDeTexto() {
8        super ("Rótulo e Campo de Texto");
9        Container c = getContentPane();
10       c.setLayout (new FlowLayout() );
11       c.add (label); c.add (texto);
12       texto.addActionListener(new Ouvinte());
13       setSize (410,100);
14       setVisible(true);
15    }
16 }
```

A classe `Ouvinte`, interna à classe `EntradaDeTexto`, implementa o tratamento de evento associado ao objeto `JTextField`. A definição da classe ouvinte como uma classe interna é interessante quando o tratador de evento tem necessidade de ter acesso a dados do componente a que está associado.

```
1  private class Ouvinte implements ActionListener {
2    public void actionPerformed(ActionEvent e) {
3      String s = "";
4      if (e.getSource() == texto)
5          s = "Olá " + e.getActionCommand() + "!";
6      JOptionPane.showMessageDialog (null, s);
7    }
8  }
```

O tratador de eventos `actionPerformed` testa na linha 4 se o objeto originador do evento é de fato a caixa `texto`, e, assim sendo,

recupera, via comando **getActionCommand**, o texto digitado na caixa, e, antes de retornar, exibe uma messagem na tela.

E o programa principal que produz o resultado da Fig. 19.3 é o seguinte:

```
1 public class UsaEntradaDeTexto {
2   public static void main (String args[]) {
3     EntradaDeTexto j = new EntradaDeTexto();
4      j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5   }
6 }
```

19.2 Classes adaptadoras

Grande parte das interfaces de ouvintes tem apenas um método para tratamento de eventos. Outras, contudo, define vários métodos, que, por serem abstratos, devem ser todos implementados pela classe do ouvinte, mesmo quando apenas alguns deles sejam de interesse. Por exemplo, a interface **WindowListener**, mostrada na Fig. 19.4, tem sete tratadores de eventos.

Interface	Métodos	Parâmetro/ Acessadores	Origem
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent getWindow	Window

Figura 19.4 Interface `WindowListener`

O que se faz nesses casos é implementar os tratadores que não interessam com corpo vazio ({ }).

Para evitar esse trabalho, Java define as chamadas classes adaptadoras, que constituem uma alternativa às interfaces ouvintes de eventos. Essas classes atribuem corpo vazio a todos os métodos das interfaces que implementam, permitindo ao programador reimplementar apenas os métodos necessários. A classe adaptadora `WindowAdapter` tem a seguinte implementação:

```
1 class WindowAdapter implements WindowListener {
2     public void windowClosing(WindowEvent e) { }
3     public void windowOpened(WindowEvent e) { }
4     public void windowIconified(WindowEvent e) { }
5     public void windowDeiconified(WindowEvent e) { }
6     public void windowClosed (WindowEvent e) { }
7     public void windowActivated (WindowEvent e) { }
8     public void windowDeactivated(WindowEvent e) { }
9 }
```

As demais classes adaptadores de eventos definidas em Java são listadas na Fig. 19.5.

Classe Adaptadora	Interface Implementada
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Figura 19.5 Classes Adaptadoras

19.3 Seleção de opções

Um *botão* é um componente em que o usuário pode clicar para acionar uma ação específica. Os botões podem ser botões de comando, como `JButton`, ou botões de estado, como `JCheckBox` e

`JRadioButton`. Botões de comando constituem um dos principais modos de interação entre o usuário e a aplicação. Botões de estado são caracterizados por valores de estado: ativado ou desativado.

19.3.1 Seleção de um botão

Um botão de comando é um componente que pode ser programado para desencadear um evento quando clicado pelo ponteiro de mouse ou acionado pelo usuário de alguma outra forma. O evento gerado é do tipo `ActionEvent`, que deve ser tratado pelo método `actionPerformed` do ouvinte que implementa a interface `ActionListener`.

Para mostrar o funcionamento de botões de comando, considere o botão definido como o da Fig. 19.6 e que um clique com o ponteiro do *mouse* nesse botão gere a saída da Fig. 19.7.

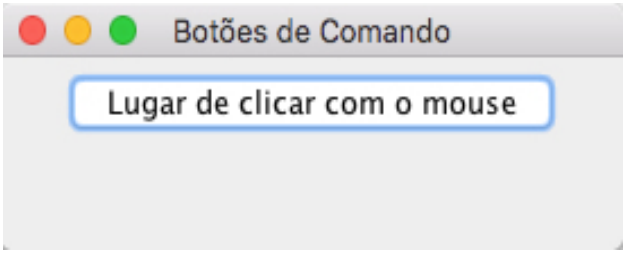


Figura 19.6 Um Botão

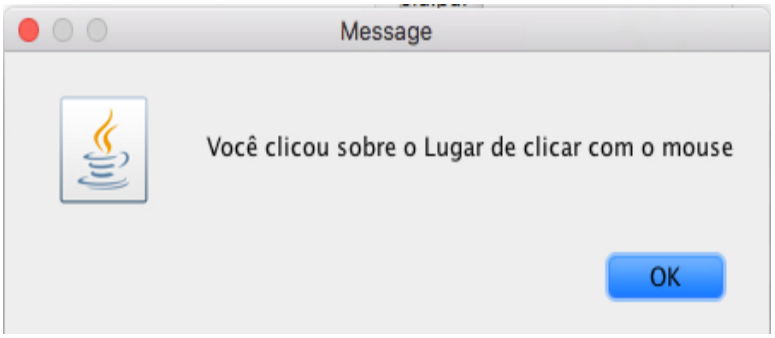


Figura 19.7 Resposta da Aplicação

O componente do tipo **JFrame** que hospeda o botão citado acima é definido pela classe **ExemploBotao1**, a qual associa-lhe um ouvinte de evento do tipo **ActionListener**, que é definida como uma classe interna.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class ExemploBotao1 extends JFrame {
5      private class Ouvinte implements ActionListener{ ... }
6      private JButton botao =
7        new JButton ("Lugar de clicar com o mouse");
8      public ExemploBotao1() {
9          super ("Botões de Comando");
10         Container c = getContentPane();
11         c.setLayout (new FlowLayout());
12         c.add (botao);
13         botao.addActionListener(new Ouvinte());
14         setSize (275,100);  setVisible(true);
15     }
16 }
```

onde destaca-se a declaração da classe do ouvinte na linha 5, a qual tem a seguinte definição:

```
1  private class Ouvinte implements ActionListener {
2    public void actionPerformed (ActionEvent e){
3      JOptionPane.showMessageDialog (null,
4        "Você clicou sobre o " + e.getActionCommand());
5      }
6  }
```

A classe **Ouvinte** tem um único método, **actionPerformed**, que é automaticamente acionado em resposta a cliques no botão ao qual o objeto ouvinte estiver associado, recupera do descritor de evento **e** que lhe for passado como parâmetro, via a operacão **getActionCommand**, o rótulo do botão que recebeu o clique.

O programa principal dessa aplicação apresentado a seguir segue o padrão usado nos exemplos anteriores.

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  public class UsaExemploBotao1 {
4    public static void main (String args[]){
5      ExemploBotao1 j = new ExemploBotao1();
6      j.addWindowListener (
7        new WindowAdapter () {
8          public void windowClosing (WindowEvent e){
9             System.exit(0);
10         }
11       });
12   }
13 }
```

19.3.2 Seleção múltipla de botões

O pacote `swing` disponibiliza vários tipos de botões de estado, dentre eles estão `JCheckBox` e `JRadioButton`.

Botões de estado geralmente são utilizados para selecionar ou desselecionar uma ou mais opções de um conjunto de possibilidades, via cliques nos componentes apropriados, que acionam o método `itemStateChanged` dos respectivos ouvintes, cuja interface é a seguinte:

```
interface ItemListener {
   void itemStateChanged(ItemEvent e);
}
```

onde o objeto do tipo `ItemEvent` passado via o parâmetro `e` pelo sistema de tratamento de eventos, quando é feita uma seleção de botões de estado ou de itens de uma lista, possui os seguintes atributos e operações:

- `ItemEvent.SELECTED`:
  indica que item foi selecionado.
- `ItemEvent.DESELECTED`:
  indica que item foi desselecionado.
- `Object getItem()`:
  retorna o item afetado pelo evento.
- `int getStateChange()`:
  retorna o tipo de mudança de estado (`SELECTED` ou `DESELECTED`).
- `ItemSelectable getItemSelectable`:
  retorna o objeto originador do evento.
- `int getStateChange`:
  retorna o tipo de mudança de estado: `SELECTED` ou `DESELECTED`.
- `String paramString()`:
  retorna a identificação do evento.

A Fig. 19.8 resume os atributos de `ItemListener` e indica alguns componentes a que ouvintes desse tipo podem ser associados.

Interface	Método	Parâmetro/ Acessadores	Origem
Item-Listener	itemSta-teChanged	ItemEvent paramString getItemSelectable getStateChange getItem	JCheckBox JList Choice

Figura 19.8 Ouvintes de Botões de Estado

Componentes que aceitam ouvintes do tipo `ItemListener` são os listados na última coluna da tabela da Fig. 19.8. O exemplo a seguir mostra como associar e tratar esse tipo de evento com o componente `JCheckBox`. A proposta é implementar uma interface que inicialmente exibe a Fig.19.9, que solicita ao usuário modificar

o estilo das letras da caixa de texto, tendo as opções de escolher `Negrito`, `Itálico`, ambos ou nenhum deles.

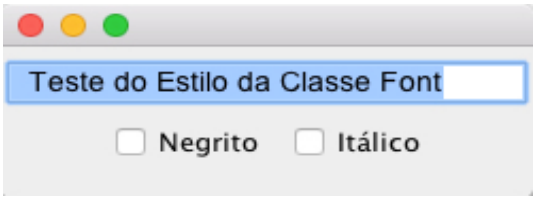


Figura 19.9 JCheckBox: Janela Inicial

As figuras 19.10, 19.11 e 19.12 mostram o efeito de cada seleção e desseleção de itens do componente `JCheckBox`.

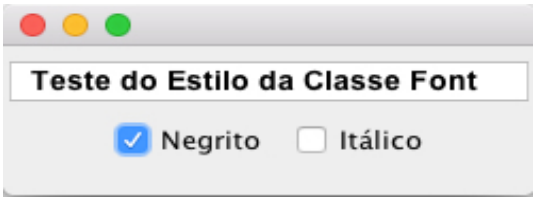


Figura 19.10 JCheckBox: Após Clique em `Negrito`

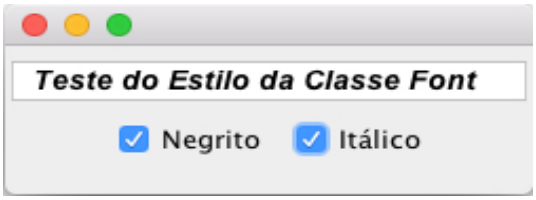


Figura 19.11 JCheckBox: Após clique `Itálico`

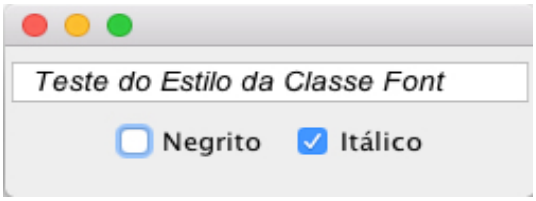


Figura 19.12 JCheckBox: Após novo clique em `Negrito`

O componente da Fig. 19.9 é definido por `DiversasOpcoes1`, que declara uma classe interna de nome `Controlador` para cons-

truir o ouvinte de eventos, e também cria os componentes do tipo `JCheckBox`, `boxItalico` e `boxNegrito`, e uma caixa de texto `t`, necessários à aplicação. Observe que a caixa de texto `t` é um `JTextField` previamente inicializado.

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  class DiversasOpcoes1 extends JFrame {
4    private JCheckBox boxNegrito = new JCheckBox("Negrito");
5    private JCheckBox boxItalico = new JCheckBox("Itálico");
6    private JTextField t  =
7      new JTextField("  Teste do Estilo da Classe Font",18);
8    private int negrito = Font.PLAIN;
9    private int italico = Font.PLAIN;
10   public DiversasOpcoes1 () {
11     Container c = getContentPane();
12     c.setLayout (new FlowLayout());
13     t.setFont (new Font ("Arial", Font.PLAIN, 14));
14     c.add(t); c.add(boxNegrito); c.add(boxItalico);
15     boxNegrito.addItemListener(new Controlador());
16     boxItalico.addItemListener(new Controlador());
17     setSize (200,100); setVisible(true);
18   }
19   private class Controlador implements ItemListener {...}
20 }
```

A construtora de `DiversasOpcoes1` segue um padrão linear de providências: adiciona os componentes criados e devidamente configurados na camada de conteúdo da janela e associa um mesmo ouvinte do tipo `Controlador` aos componentes `boxNegrito` e `boxItalico`. O uso de um mesmo ouvinte é possivel porque o tratador de eventos `itemStateChanged` é capaz de identificar o originador do evento.

O ouvinte associado a `boxNegrito` e `boxItalico`, quando acionado em reação a cliques nesses componentes, dispara a execução

do método **itemStateChanged** da classe **Controlador**, apresentada abaixo, com um parâmetro do tipo **ItemEvent**, que descreve o evento gerado. O papel desse tratador de eventos é mudar o fonte da caixa de texto conforme solicitado.

```
1  private class Controlador implements ItemListener {
2     public void itemStateChanged(ItemEvent e) {
3        if (e.getSource() == boxNegrito)
4            if (e.getStateChange() == ItemEvent.SELECTED)
5                 negrito = Font.BOLD;
6            else negrito = Font.PLAIN;
7        if (e.getSource() == boxItalico)
8            if (e.getStateChange() == ItemEvent.SELECTED)
9                 italico = Font.ITALIC;
10           else italico = Font.PLAIN;
11       t.setFont(new Font("Arial",negrito + italico,14));
12       t.repaint();
13    }
14 }
```

Finalmente o programa principal padrão:

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  public class UsaDiversasOpcoes1 {
4    public static void main (String args[]) {
5      DiversasOpcoes1 j = new DiversasOpcoes1();
6      j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7    }
8  }
```

19.3.3 Seleção exclusiva de botões

Os componentes **JRadioButton** e **JButtonGroup** são usados quando deseja-se oferecer ao usuário um conjunto de opções dentre as

quais somente uma pode ser escolhida de cada vez, isto é, a seleção de uma opção desliga a opção selecionada anteriormente.

Cada um dos botões `JRadioButton` que mutuamente se excluem deve ser adicionado a uma instância da classe `ButtonGroup`. Assim, quando um dos botões de um mesmo grupo é clicado, ele é marcado como selecionado e o que estava marcado pela interação anterior é automaticamente desselecionado, e um evento é gerado, provocando o acionamento do ouvinte associado ao botão clicado.

Para mostrar como implementar esses componentes, o exemplo de seleção de estilo baseado em `JCheckBox` é agora modificado para operar com seleções exclusivas. A janela inicial do programa `UsaUmaOpção1` é a da Fig. 19.13.

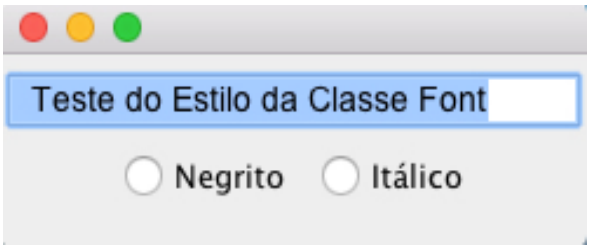


Figura 19.13 ButtonGroup: Janela Inicial

Após clique no botão `Negrito`, a caixa de texto tem o fonte mudado para negrito, e o clique em `Italico`, aplicado a seguir, retira o negrito e muda o fonte para itálico, como mostrado na Fig. 19.14.

Figura 19.14 ButtonGroup: Após Cliques em Negrito e Italo

A classe `UmaOpcão1` que implementa o componente principal

do programa **UsaUmaOpção1** é apresentada a seguir, onde merece destaque a criação do grupo de botões exclusivos, do tipo **JButtonGroup**, na linha 8 e seu preenchimento nas linhas 19 e 20.

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  class UmaOpção1 extends JFrame {
4    private JRadioButton boxNegrito =
5                              new JRadioButton("Negrito");
6    private JRadioButton boxItalico =
7                              new JRadioButton("Itálico");
8    private ButtonGroup grupo = new ButtonGroup();
9    private JTextField t  =
10     new JTextField("  Teste do Estilo da Classe Font",20);
11   private int estilo = Font.PLAIN;
12   public UmaOpção1 () {
13      Container c = getContentPane();
14      c.setLayout(new FlowLayout());
15      t.setFont (new Font("Arial",estilo, 14));
16      c.add(t);
17      c.add(boxNegrito);
18      c.add(boxItalico);
19      grupo.add(boxNegrito);
20      grupo.add(boxItalico);
21      boxNegrito.addItemListener(new Controlador());
22      boxItalico.addItemListener(new Controlador());
23      setSize (250,100);
24      setVisible(true);
25   }
26   private class Controlador implements ItemListener {...}
27 }
```

A classe interna **Controlador** implementa o método tratador de eventos **itemStateChanged** do ouvinte dos botões de rádio, conforme definido na classe **UmaOpção1** apresentada acima. Esse tratador apenas identifica o originador do evento e troca o estilo

do fonte da caixa de texto, conforme solicitado. As operações de remover a marca visual de seleção de um botão e de marcação do seguinte são realizadas automaticamente.

```
1  private class Controlador implements ItemListener {
2     public void itemStateChanged(ItemEvent e) {
3        if (e.getSource() == boxNegrito)
4            estilo = Font.BOLD;
5        if (e.getSource() == boxItalico)
6            estilo = Font.ITALIC;
7        t.setFont(new Font("Arial", estilo, 14));
8        t.repaint();
9     }
10 }
```

E o programa principal padrão dos exemplos:

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class UsaUmaOpção1 {
5     public static void main(String args[]) {
6        UmaOpção1 j = new UmaOpção1();
7        j.addWindowListener (
8            new WindowAdapter () {
9                public void windowClosing(WindowEvent e) {
10                  System.exit(0);
11               }
12           }
13       );
14    }
15 }
```

19.3.4 Seleção em caixas de combinação

Uma caixa de combinação é o `JComboBox`, que combina um item com uma lista *drop-down*, da qual o usuário pode escolher um item

por meio de um clique que gera um evento percebido por ouvintes do tipo `ItemListener`.

Os itens da lista são identificados por índices, sendo 0 (zero) o primeiro. O primeiro item adicionado à lista aparece como item selecionado quando `JComboBox` for exibida e para cada seleção são gerados dois eventos: um para o item desselecionado e outro para o selecionado.

Resumidamente, o protocolo de criação de uma caixa de combinação desse tipo segue os passos do seguinte modelo:

```
private String nomes[] = {"item1","item2,"item3"};
Container c = getContentPane();
JComboBox<String> caixa = new JComboBox<String>(nomes);
caixa.setMaximumRowCount(12);
caixa.addItemListener(new Ouvinte());
c.add(caixa);
```

E o ouvinte do componente `JComboBox` deve implementar a interface `ItemListener`. As principais operações de `JComboBox<E>` são as seguintes:

- `JComboBox<E>(E[] nomes)`:
  constrói uma caixa de combinação com os itens do tipo `E`, definidos pelo arranjo `nomes`. Os nomes passados são convertidos pela operação `toString`.
- `setMaximumRowCount(int n)`:
  define o número de linhas visíveis da caixa de combinação. Se necessário, barras de rolagem serão incluídas.
- `int getSelectedIndex()`:
  índice dos itens inicia-se com 0. Usado pelo ouvinte para saber qual foi o item selecionado pelo clique.
- `void addItemListener(ItemListener ouvinte())`:
  adiciona o ouvinte de eventos ao componente.

A seguinte aplicação usa um componente do tipo `JComboBox` para implementar um menu de escolha de vinhos de uma lista, por

exemplo, merlot, cabernet, sirah, carménère e chardonnay. Inicialmente, a aplicação exibe a pequena janela da Fig. 19.15, onde uma seleção prévia já foi realizada.

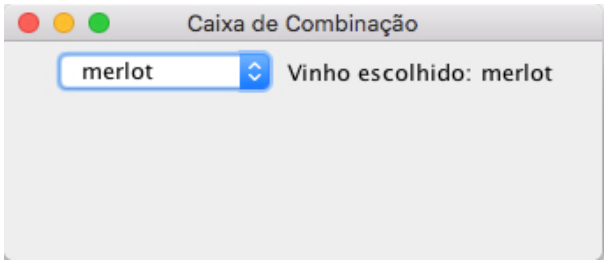


Figura 19.15 JComboBox: Janela Inicial

Após um clique no circunflexo no canto direito do componente, o `JComboBox` expande-se conforme a Fig. 19.16,

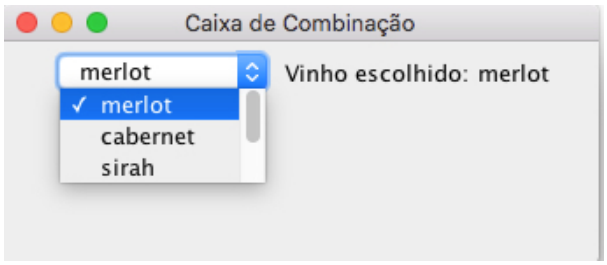


Figura 19.16 JComboBox: Itens no Início

a qual exibe os três primeiros vinhos da lista com a marcação de que `merlot` é o selecionado por *default* e uma barra de rolagem para visualizar os demais itens da lista.

Fig. 19.17 mostra que a barra de rolagem foi deslizada para baixo, e o item `chardonnay` selecionado.

A classe `Vinhos1` implementa a caixa de combinação descrita acima, alocando na camada de conteúdo da janela exibida três componentes: um `JComboBox`, para selecionar vinhos, e dois `JLabel`, para mostrar o resultado da seleção. E para tratar os eventos, cria o ouvinte cujo tipo `ItemListener` foi declarado internamente à classe `Vinhos1` e o associa ao componente `caixa`.

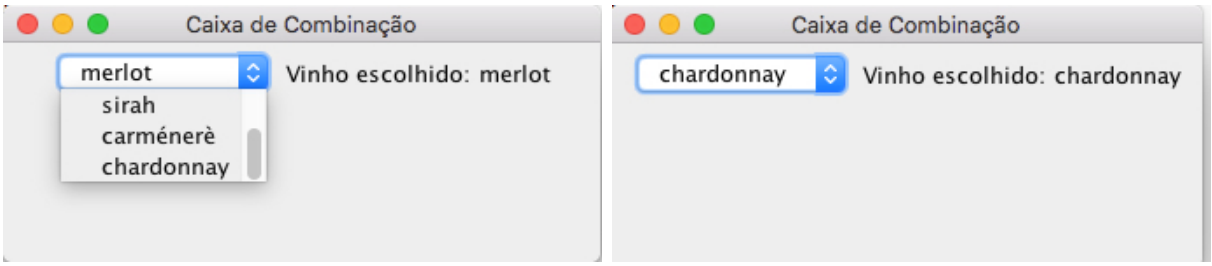


Figura 19.17 JComboBox: Chardonnay Selecionado

```
1  import java.awt.event.*; import java.awt.*;
2  import javax.swing.*;
3  public class Vinhos1 extends JFrame {
4     private String uvas[]  = { "merlot", "cabernet",
5                        "sirah", "carmenere", "chardonnay" };
6     private JComboBox<String> caixa =
7                                new JComboBox<String>(uvas);
8     private JLabel escolha = new JLabel(uvas[0]);
9     private JLabel rotulo = new JLabel("Vinho escolhido:");
10    public Vinhos1() {
11       super("Teste de Caixa de Combinacao" );
12       Container c = getContentPane();
13       c.setLayout(new FlowLayout());
14       caixa.addItemListener(new Ouvinte());
15       caixa.setMaximumRowCount(3);
16       c.add(caixa);  c.add(rotulo); c.add(escolha);
17       setSize(340,150); setVisible(true);
18    }
19
20    private class Ouvinte implements ItemListener {
21       public void itemStateChanged( ItemEvent event ) {
22         if (event.getStateChange() == ItemEvent.SELECTED)
23           escolha.setText(uvas[caixa.getSelectedIndex()]);
24       }
25    }
26 }
```

E o programa principal padrão para o presente exemplo:

```
1 import javax.swing.*;
2 public class UsaVinhos1 {
3     public static void main (String args[]) {
4         JFrame j = new Vinhos1();
5         j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6     }
7 }
```

19.3.5 Seleção simples em listas

Caixas do tipo `Jlist` permitem a seleção de um ou mais itens de uma lista. As principais operações do componente `JList<E>`, onde o parâmetro `<E>` designa o tipo dos elementos exibidos na lista, são os seguintes:

- `JList()`:
  constrói lista vazia.
- `JList<E>(E[] itens)`:
  contrói uma `JList` que exibe os elementos do arranjo `itens`, usualmente convertido via `toString`.
- `void setVisibleRowCount(int n)`:
  define propriedades das linhas visíveis, i.e., o número de linhas visíveis sem necessidade de deslizamento da barra de rolagem.
- `void setSelectionMode(int mode)`:
  define o modo de seleção de itens, que pode ser um dos inteiros:
  - `ListSelectionModel.SINGLE_SELECTION`: permite selecionar um único item de cada vez.
  - `ListSelectionModel.SINGLE_INTERVAL_SELECTION`: permite selecionar um intervalo contíguo de itens, com o auxílio da teclas *Shift*.
  - `ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`: permite selecionar itens não-contíguos com o auxílio da tecla *Command*.

- `void setListData(E[] itens)`:
  configura os itens dados, por default convertidos com `toString`, no `JList` corrente.
- `int getSelectedIndex()`:
  informa o item de menor índice selecionado da lista.
- `int[] getSelectedIndices()`:
  retorna todos os índices selecionados em ordem crescente.
- `E getSelectedValue()`:
  retorna o rótulo do item selecionado.
- `List<E> getSelectedValuesList()`:
  retorna os rótulos dos itens selecionados.
- `void setFixedCellWidth(int w)`:
  define a largura da célula de cada item na lista.
- `void setFixedCellHeight(int h)`:
  define a altura da célula de cada item na lista.

Figura 19.18 JList: Início da Lista

O primeiro exemplo de uso de `JList`, o programa `Autoria1`, é uma aplicação que permite selecionar um item de cada vez de uma lista de títulos de livros e mostrar em um `JTextField` o nome do autor do livro selecionado. Na abertura do programa, a janela da Fig. 19.18 é exibida, onde pode-se observar a presença da barra de rolagem, que foi incorporada no componente, porque há mais

elementos na lista do que cabe na janela de exibição.

A Fig. 19.19 é o resultado do deslizamento da lista para baixo, arrastando a barra de rolagem com o ponteiro do *mouse*. O uso de barras de rolagem é necessário sempre que não for possível prever com exatidão quantos itens a lista terá, e também para reduzir o tamanho do componente pela exibição somente parcial de sua lista.

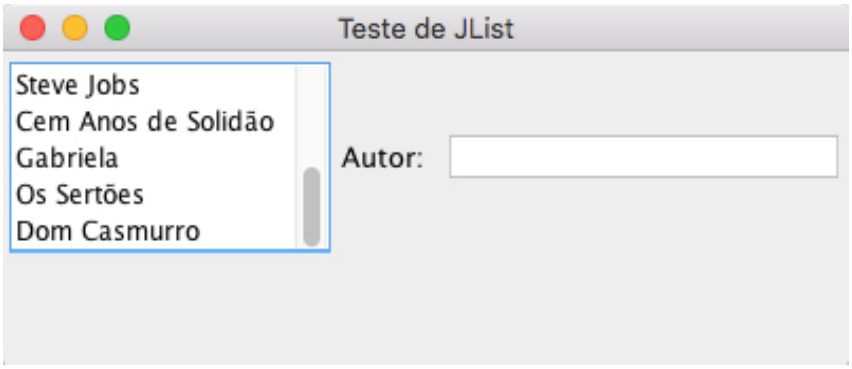


Figura 19.19 JList: Rolagem até o fim da Lista

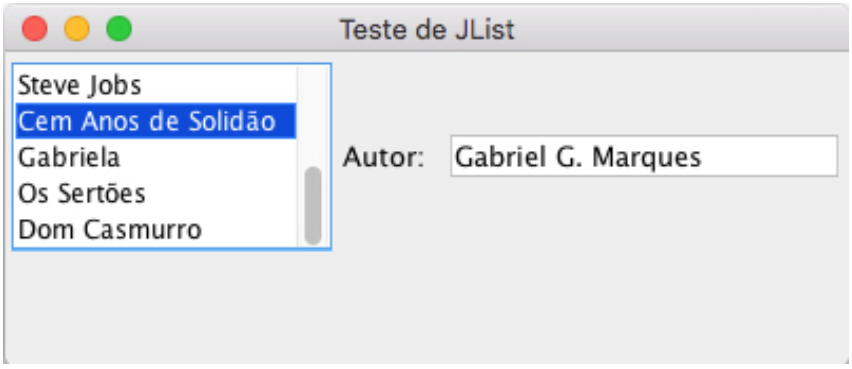


Figura 19.20 JList: Selecionado Cem Anos de Solidão

E um clique no item "`Cem Anos de Solidão`"da caixa de títulos tem o efeito de marcar o item selecionado na lista e escrever o nome do autor do livro escolhido na caixa de texto, conforme mostra a Fig. 19.20.

A classe `Autoria1` implementa a caixa de seleção exibida nas figuras Fig. 19.18 a 19.20.

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.event.*;
4  class Autoria1 extends JFrame {
5     private final String obras[] = {
6        "Lusíadas", "Don Quixote", "Teoria da Relatividade",
7        "Linguagem Java", "Big Bang", "Grande Sertão",
8        "Steve Jobs", "Cem Anos de Solidão", "Gabriela",
9        "Os Sertões", "Dom Casmurro" };
10    private final String autores[] = {
11       "Camões",  "Miguel de Cervantes", "Einstein",
12       "Deitel",  "Simon Sign", "Guimarães Rosa",
13       "Walter Isaacson", "Gabriel G. Marques",
14       "Jorge Amado", "Euclides da Cunha",
15       "Machado de Assis"};
16    private JList<String> lista;
17    private JLabel rotulo = new JLabel("Autor: ");
18    private JTextField autor  = new JTextField(15);
19    private Container c = getContentPane();
20    private class Ouvinte
21                   implements ListSelectionListener {...}
22    public Autoria1() {
23       super("Teste de JList" );
24       c.setLayout(new FlowLayout() );
25       lista = new JList<String>(obras);
26       lista.setVisibleRowCount(5);
27       lista.setSelectionMode(
28                     ListSelectionModel.SINGLE_SELECTION );
29       c.add(new JScrollPane(lista));
30       lista.addListSelectionListener(new Ouvinte());
31       c.add(rotulo); c.add(autor);
32       setSize(400,170);  setVisible(true);
33    }
34 }
```

A construtora de **Autoria1** cria os três componentes da janela,

um `JList`, um `JLabel` e um `JTexField`, e associa um ouvinte de eventos ao componente `JList`.

Três pontos na classe `Autoria1` merecem destaque. O primeiro é o comando da linha 27, o qual restringe a seleção na caixa de combinação a apenas um item de cada vez. O segundo ponto trata do uso da classe `JScrollPane` na linha 29 para introduzir barras de rolagem. E o terceiro ponto, na linha 20, é a declaração da classe `Ouvinte` como interna à classe do componente `Autoria1`. Isso é feito para facilitar o acesso aos campos `autor` e `lista` do originador do evento, como pode ser visto a seguir.

```
1 private class Ouvinte implements ListSelectionListener {
2    public void valueChanged(ListSelectionEvent event) {
3       autor.setText(autores[lista.getSelectedIndex()]);
4    }
5 }
```

E o programa principal:

```
1 import javax.swing.JFrame;
2 public class UsaAutoria1 {
3  public static void main( String args[]) {
4     JFrame lista = new Autoria1();
5     lista.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6  }
7 }
```

19.3.6 Configuração de itens de listas

A construtora `JList(E[] itens)` ou o método `setListData` automaticamente chamam o método `toString` para converter os itens do tipo parametrizado `E` para um formato legível pelo usuário, normalmente usando o método `toString`.

Alternativamente, pode-se dar aos elementos de suas listas uma outra aparência via o método

`void setCellRenderer(ListCellRenderer renderer)`

de `JList`, o qual delega a

`renderer.getListCellRendererComponent`,

a configuração de cada um dos itens da `JList`.

Os itens com a configuração desejada são obtidos pelo método da interface `ListCellRenderer`:

```
public Component getListCellRendererComponent(
       JList list, Object value, int index,
       boolean isSelected, boolean cellHasFocus);
```

onde

- `JList list`: o componente `JList` que está sendo configurado.
- `Object value`: o valor do item a ser exibido.
- `int index`: índice do item dentro da lista.
- `boolean isSelected`: indica se o item está selecionado.
- `boolean cellHasFocus`: verdadeiro se o item tem foco.

Figura 19.21 JList com Itens Configurados

O exemplo a seguir, `UsaAuditoria2`, mostra uma `JList` de botões coloridos, que, após seleção de "`Cem Anos de Solidão`", produz o resultado exibido na Fig. 19.21.

Na implementação abaixo da interface `ListCellRenderer`, o parâmetro `value` da função de configuração é transformado em botão de fundo colorido e rotulado com seu valor.

```
1  import java.awt.*; import javax.swing.*;
2  class Configuração implements ListCellRenderer {
3     private Color[] colors = {Color.red, Color.blue,
4       Color.cyan, Color.yellow, Color.pink, Color.green,
5       Color.lightGray, Color.magenta, Color.orange,
6       Color.gray, Color.darkGray, Color.white, Color.black
7     };
8     public Component getListCellRendererComponent(
9             JList list, Object value,  int index,
10            boolean isSelected, boolean cellHasFocus) {
11       index = index%colors.length;
12       JButton item = new JButton();
13       item.setText(value.toString());
14       item.setOpaque(true);
15       item.setBackground(colors[index]);
16       return item;
17    }
18 }
```

A classe `Autoria2` é essencialmente igual a `Autoria1`, exceto por alguns detalhes. Em primeiro lugar, as linhas 3 e 4 da nova classe interna `Ouvinte`, escrevem o nome da obra na caixa de texto de saída, pois, com a configuração de itens implementada, a marcação do item escolhido deixou de ser visível.

```
1  private class Ouvinte implements ListSelectionListener {
2     public void valueChanged(ListSelectionEvent event) {
3        int i = lista.getSelectedIndex();
4        autor.setText(obras[i] + ": " + autores[i]);
5     }
6  }
```

E, na linha 27 de **Autoria2**, a classe **Configuração** é usada para redefinir o método de configuração de itens da **lista**.

```
1  import java.awt.*;import javax.swing.*;
2  import javax.swing.event.*;
3  class Autoria2 extends JFrame {
4     private final String obras[] = {
5      "Lusíadas", "Don Quixote", "Teoria da Relatividade",
6      "Linguagem Java", "Big Bang", "Grande Sertão",
7      "Steve Jobs", "Cem Anos de Solidão", "Gabriela",
8      "Os Sertões", "Dom Casmurro" };
9     private final String autores[] = {
10     "Camões", "Miguel de Cervantes", "Einstein",
11     "Deitel", "Simon Sign", "Guimarães Rosa",
12     "Walter Isaacson", "Gabriel G. Marques",
13     "Jorge Amado","Euclides da Cunha","Machado de Assis"};
14    private JList<String> lista;
15    private JLabel rotulo = new JLabel("Autor: ");
16    private JTextField autor  = new JTextField(25);
17    private Container c = getContentPane();
18    private class Ouvinte
19            implements ListSelectionListener {...}
20    public Autoria2() {
21       super("Teste de JList" );
22       c.setLayout(new FlowLayout() );
23       lista = new JList<String>(obras);
24       lista.setVisibleRowCount(5);
25       lista.setSelectionMode(
26             ListSelectionModel.SINGLE_SELECTION );
27       lista.setCellRenderer(new Configuracao());
28       c.add(new JScrollPane(lista));
29       lista.addListSelectionListener(new Ouvinte());
30       c.add(rotulo); c.add(autor);
31       setSize(700,200);  setVisible(true);
32    }
33 }
```

O programa `Auditoria2` segue o padrão dos exemplos.

```
1 import javax.swing.JFrame;
2 public class UsaAutoria2 {
3   public static void main( String args[]) {
4     JFrame lista = new Autoria2();
5     lista.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6   }
7 }
```

19.3.7 Seleção múltipla em lista

O segundo exemplo de `JLIST` é o programa `UsaCompras1`, que é um sistema que permite ao comprador selecionar vários livros de uma lista e transferi-los para o carrinho de compras. A janela gráfica tem três componentes: um `JLIST`, que é o repositório de livros, um `JButton`, que é o comando de compra, e um segundo `JList`, para exibir o resultado da compra.

A Fig. 19.22 é a tela inicial, que exibe uma lista de livros, dos quais pode-se selecionar um ou mais, e compra é efetivada por um clique no botão `Compre`, que tem o efeito de colocar no *carrinho* à direita a seleção feita. Para simplificar o exemplo, o botão `Compre` não permite acrescentar novas compras ao carrinho. Sempre que for clicado, a lista de compra é reiniciada com os itens marcados.

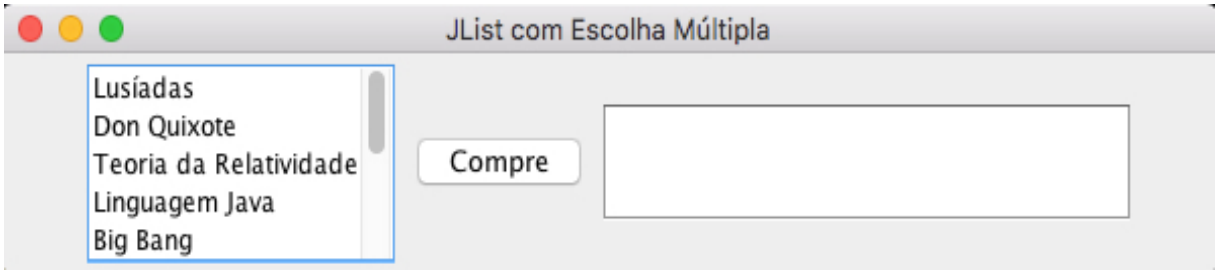


Figura 19.22 JList: Início da Compra

A Fig. 19.23 é o resultado da seleção múltipla de itens contíguos, que inicia com a seleção de "`Dom Quixote`"e, com a tecla *Shift* pressionada, faz-se a seleção de "`Linguagem Java`". Essa operação seleciona os itens do primeiro clique ao do segundo. Um clique no botão `Compre` causa o preenchimento da lista à direita.

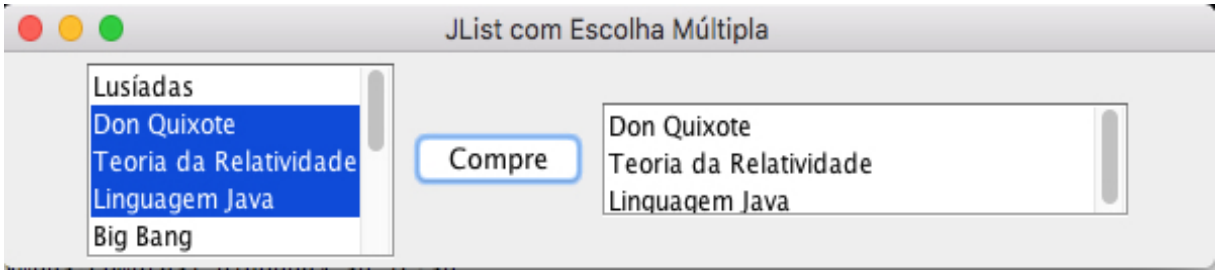


Figura 19.23 JList: Seleção de 3 Livros via Tecla *SHIFT*

A Fig. 19.24 mostra uma segunda compra onde são selecionados itens múltiplos não-contíguos, por meio da tecla *Command*, que é mantida pressionada durante marcação dos itens desejados.

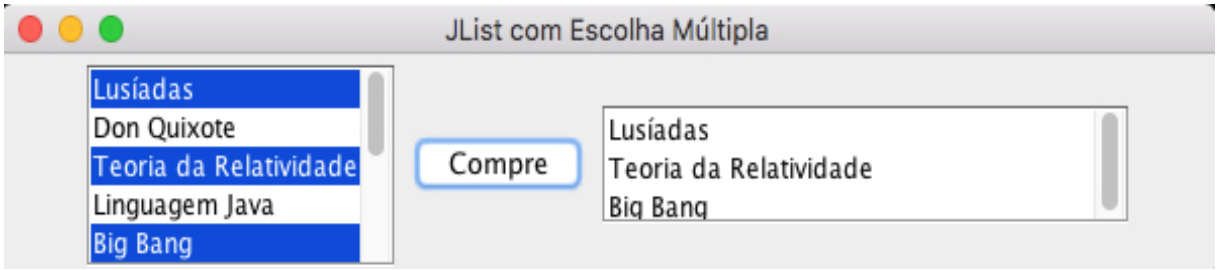


Figura 19.24 JList: Seleção de 3 Livros via Tecla *Command*

A classe central da aplicação, `Compras1`, implementa as janela de compras descritas nas Figuras 19.22, 19.23 e 19.24. Nessa classe, estão definidos a classe interna `Ouvinte`, que cuida do tratamento de eventos de botão, e os componentes `livros` do tipo `JList`, `carrinho` do tipo `JButton` e `comando` do tipo `JList`.

A classe `Ouvinte` deve ser interna pois faz acesso aos componentes `livros` e `carrinho`.

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*; import java.util.List;
3  class Compras1 extends JFrame {
4     private final String nomes[] = {
5        "Lusíadas", "Don Quixote", "Teoria da Relatividade",
6        "Linguagem Java", "Big Bang", "Grande Sertão",
7        "Steve Jobs", "Cem Anos de Solidão", "Gabriela",
8        "Os Sertões", "Dom Casmurro" };
9     private JList<String> livros = new JList<String>(nomes);
10    private JButton comando = new JButton("Compre");
11    private JList<String> carrinho = new JList<String>();
12    private Container c = getContentPane();
13    class Ouvinte implements ActionListener {...}
14    public Compras1() {...}
15 }
```

Na função construtora de **Compras1**, apresentada a seguir, destacam-se o comando das linhas 5 e 6, que habilita a **JList livros** a aceitar seleção múltipla, e os comandos das linhas 7 e 11, que instalam barras de rolagem em ambos os componentes **JList**.

```
1  public Compras1() {
2     super("JList com Escolha Múltipla" );
3     c.setLayout(new FlowLayout());
4     livros.setVisibleRowCount(5);
5     livros.setSelectionMode(
6           ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
7     c.add(new JScrollPane(livros));
8     comando.addActionListener(new Ouvinte());
9     c.add(comando);
10    carrinho.setVisibleRowCount(3);
11    c.add(new JScrollPane(carrinho));
12    setSize(600,120);
13    setVisible(true);
14 }
```

A classe `Ouvinte` implementa o efeito de clique no botão `Compre`, isto é, transfere os livros selecionados para o carrinho de compras. Note que, a cada ativação do tratador de eventos `actionPerformed`, o carrinho de compras é reinicializado pelo método `setListData`.

```
1  class Ouvinte implements ActionListener {
2     public void actionPerformed(ActionEvent e) {
3       List<String> selecao =
4                             livros.getSelectedValuesList();
5       Object[] a = selecao.toArray();
6       String[] s = new String[s1.length];
7       for (int i=0; i<s1.length ; i++) s[i] = (String)a[i];
8       carrinho.setListData(s2);
9     }
10 }
```

O programa principal `UsaCompras1` segue o padrão adotado nos exemplos anteriores de simplesmente criar um componente `JFrame`, que pode ser fechado com um clique no botão `x` ou vermelho:

```
1  import javax.swing.JFrame;
2  public class UsaCompras1 {
3    public static void main( String args[]) {
4      JFrame Compras1 = new Compras1();
5      compras.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6    }
7  }
```

19.4 Conclusão

Os recursos para programação de interfaces gráficas em Java é extremamente rico com muitas dezenas de classes e centenas de métodos. Neste capítulo, mais alguns componentes importantes

foram abordados. Para aprofundar o assunto, recomenda-se a leitura dos próximos capítulos e da bibliografia arrolada, em especial os livros dos Deitels.

Exercícios

1. Qual é a utilidade de classes adaptadoras?

2. Descreva uma aplicação em que um evento, por exemplo, o clique do *mouse* sobre um botão, deve ser tratado por muitos ouvintes.

Notas bibliográficas

Um dos melhores texto para estudar e aprofundar programação de interfaces gráfica é o livro *Java - Como programar* dos Deitels[10]. Os capítulos 11, 12 e 22 desse livro apresentam muitos exemplos ilustrativos de uso dos pacotes `Swing` e `AWT`. Vale a pena ler.

Capítulo 20

GUI: Organização de Componentes

Uma janela com muitos componentes gráficos pode tornar o posicionamento desses elementos inadministrável. A arrumação dos objetos em uma janela pode ser facilitada pela sua organização em grupos de componentes relacionados e se cada grupo, por sua vez, puder ser tratado como um componente unitário, que internamente tem seu próprio leiute, permitindo assim uma organização hierárquica.

O mecanismo para criar esses agrupamentos de componentes é a classe `JPanel` do pacote `javax.sing`. Objetos dessa classe são chamados de painéis e são usados como uma base onde os componentes gráficos podem ser fixados, inclusive outros painéis.

20.1 Classe `JPanel`

Painéis têm um comportamento muito próximo ao de objetos da classe `JFrame`, pois a classes dos painéis e a de janelas estão na hierarquia da classe `java.awt.Container`. A classe `JPanel` é uma extensão direta de `JComponent`.

A rigor, uma janela `JFrame` possui quatro camadas, também chamadas de *vidraças*, denominadas *Glass Pane*, *Content Pane*, *Layered Pane* e *Root Pane*, que são acessíveis via operações de `JFrame` de nomes `get`vidraça`Pane`, onde vidraça pode ser subs-

tituída por `Glass`, `Content`, `Layered` ou `Root`. A vidraça *Content Pane* é um objeto do tipo `Container`, assim como são os objetos do tipo `JPanel`, o que explica a similaridade de tratamento desses dois componentes, que têm em comum pelo menos as seguintes operações:

- `JPanel()`:
  cria um painel com leiate `FlowLayout`.
- `JPanel(LayoutManager m)`:
  cria painel com o leiaute especificado por `m`.
- `Component add(Component c)`:
  adiciona o componente `c` no fim do contêiner.
- `void paintComponents(Graphics g)`:
  desenha todos os componentes descritos em `g`.
- `void remove(Component c)`:
  remove o component `c` do contêiner
- `void setLayout(Layoutmanager m)`:
  define o leiaute do contêiner.
- `void setSize(int largura, int altura)`:
  define o tamanho do contêiner.
- `void setFont(Font f)`:
  define o fonte dos textos do contêiner
- `void setBackground(Color c)`:
  define a cor de fundo do contiêner.
- `void setForeground(Color c)`:
  define a cor de frente do contiêner.
- `void setVisible(boolean b)`:
  exibe ou esconde o componente.

Observe que o método `paint(Graphics g)` serve para desenhar no `JFrame` e o método `paintComponent(Graphics g)` para desenhar no `JPanel`. Este método, em analogia com o `paint`

de `JFrame`, não deve ser chamado diretamente. Deve-se usar `repaint()`. E o primeiro comando de `paintComponent` deve ser sempre a chamada `super.paintComponent(g)`, que repassa o objeto gráfico `g` recebido como parâmetro para a construtora de `JComponent`.

Painéis são criados estendendo a classe `JPanel`, normalmente conforme o seguinte protocolo:

- declara-se uma classe que estende `JPanel`;
- sobrepõe-se o método `paintComponent(Graphics g)`, se for necessário desenhar no painel;
- adicionam-se componentes ao painel com `add(Component c)`.

O programa `UsaPainéis`, descrito a seguir, exemplifica o uso de `JPanel`, alocando em uma janela seis painéis, construídos a partir de janelas apresentadas no Capítulo 19, de forma a produzir a configuração mostrada na Fig. 20.1.

Figura 20.1 Uma Janela com 6 Painéis

O programa `UsaPainéis` cria uma janela definida pela classe

`Painéis`, que estende `JFrame`, a qual merece uma atenção especial, pois é a reponsável pela alocação dos seis painéis na janela principal da aplicação. Para simplificar o exemplo, todos os seis painéis usam o mesmo leiaute do tipo `FlowLayout` e a janela principal é um `GridLayout(3,2)`.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class Painéis extends JFrame {
5      private JPanel[] painéis = new JPanel[6];
6      public Painéis () {
7          super ("Demonstrando Painéis");
8          Container c = getContentPane();
9          c.setLayout(new GridLayout(3,2,5,5));
10         for (int i = 0; i < 6; i++) {
11            painéis[i] = new JPanel();
12            painéis[i].setLayout(new FlowLayout());
13         }
14         painéis[0].add(new Autoria3());
15         painéis[0].setBackground(Color.red);
16         painéis[1].add(new ExemploBotao2());
17         painéis[1].setBackground(Color.blue);
18         painéis[2].add(new DiversasOpcoes2());
19         painéis[2].setBackground(Color.pink);
20         painéis[3].add(new UmaOpção2());
21         painéis[3].setBackground(Color.black);
22         painéis[4].add(new Vinhos2();
23         painéis[4].setBackground(Color.yellow);
24         painéis[5].add(new Compras2());
25         painéis[5].setBackground(Color.magenta);
26         for (int i = 0; i < 6; i++) c.add (painéis[i]);
27         setSize (650, 400);
28         setVisible(true);
29     }
30 }
```

E o programa **UsaPainéis** segue o padrão usado até aqui, criando uma janela do tipo **Painéis** e associando-lhe um tratador padrão de eventos ao seu botão **x**.

```
1 import javax.swing.JFrame;
2 public class UsaPainéis {
3    public static void main( String args[]) {
4       JFrame j = new Painéis();
5       j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6    }
7 }
```

As classes **Autoria3**, **ExemploBotao2**, **Compras2**, **Vinhos2**, **DiversasOpcoes2** e **UmaOpção2**, usadas na classe **Painéis** são detalhadas, uma a uma, nas implementações a seguir. Para simplificar o exemplo e facilitar comparações das soluções, essas classes são cópias das classes **Autoria1**, **ExemploBotao1**, **Compras1**, **Vinhos1**, **DiversasOpcoes1** e **UmaOpção1**, respectivamente, apresentadas no Capítulo 19, onde estão declaradas como extensões de **JFrame**. Neste capítulo, as cópias correspondentes são definidas como extensões da classe **JPanel**.

O primeiro painel exibido na Fig. 20.1, o que está colocado na primeira linha do lado esquerdo da janela, é definido pela classe **Autoria3** apresentada na próxima página.

Observe que a classe **Autoria3** diferencia-se **Auditoria1** apenas pelo fato de ela estender **JPanel**, no lugar de **JFrame**, mas as operações aplicáveis a seus objetos são praticamente as mesmas, dentre as quais destacam-se as operações **setLayout** e **add** para definir leiaute e adicionar-lhe componentes. As outras as operações de anexação dos componentes ao painel, de inclusão de barras de rolagem, definição de seu tamanho e visibilidade são idênticas às de um objeto **JFrame**.

```
1  import java.awt.*;
2  import javax.swing.*;
3  import javax.swing.event.*;
4  class Autoria3 extends JPanel {
5     private final String nomes[] = {
6        "Lusíadas", "Don Quixote", "Relatividade",
7        "Linguagem Java", "Big Bang", "Grande Sertão",
8        "Steve Jobs", "Sapiens", "Gabriela",
9        "Os Sertões", "Dom Casmurro" };
10    private final String autores[] = {
11       "Camões",  "Miguel de Cervantes", "Einstein",
12       "Deitel",  "Simon Sign", "Guimarães Rosa",
13       "Walter Isaacson", "Y. N. Harari",
14       "Jorge Amado","Euclides da Cunha","Machado de Assis"};
15    private JList<String> lista;
16    private JLabel rotulo = new JLabel("Autor: ");
17    private JTextField autor  = new JTextField(11);
18    private class Ouvinte implements ListSelectionListener {
19       public void valueChanged(ListSelectionEvent event) {
20          autor.setText(autores[lista.getSelectedIndex()]);
21       }
22    }
23    public Autoria3() {
24       setLayout(new FlowLayout());
25       lista = new JList<String>(nomes);
26       lista.setVisibleRowCount(5);
27       lista.setSelectionMode(
28                ListSelectionModel.SINGLE_SELECTION );
29       add(new JScrollPane(lista));
30       lista.addListSelectionListener(new Ouvinte());
31       add(rotulo);
32       add(autor);
33       setSize(500,170);
34       setVisible(true);
35    }
36 }
```

O segundo painel, colocado à direita na primeira linha da janela da Fig. 20.1, é definido pela classe **ExemploBotao2**, que também tem uma implementação simples e direta: apenas monta um painel no qual coloca um botão que solicita ser clicado e programa o ouvinte de eventos a ele associado, o qual responde exibindo uma janela informando que o botão foi clicado pelo usuário. A cor azul de fundo foi definida na classe **Painéis**.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  class ExemploBotao2 extends JPanel {
5      private JButton botao =
6           new JButton ("Lugar de clicar com o mouse");
7
8      public ExemploBotao2() {
9          setLayout(new FlowLayout() );
10         add(botao);
11         botao.addActionListener (new Ouvinte());
12         setSize(275,100);
13         setVisible(true);
14     }
15
16     private class Ouvinte implements ActionListener {
17         public void actionPerformed (ActionEvent e) {
18             JOptionPane.showMessageDialog (null,
19                 "Voce clicou sobre o " +
20                                 e.getActionCommand() );
21         }
22     }
23
24 }
```

O terceiro painel, do lado esquerdo na segunda linha da mesma figura, é definido pela classe **DiversasOpcoes2**, que também segue o modelo padrão implementação de caixas **JCombobox** de seleção

não-exclusiva de dois botões `Negrito` e `Itálico`, que definem o estilo do texto.

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  class DiversasOpcoes2 extends JPanel {
4     private JCheckBox boxNegrito = new JCheckBox("Negrito");
5     private JCheckBox boxItalico = new JCheckBox("Itálico");
6     private JTextField t  =
7           new JTextField("  Teste do Estilo",10);
8     private int negrito = Font.PLAIN;
9     private int italico = Font.PLAIN;
10    public DiversasOpcoes2 () {
11       setLayout(new FlowLayout());
12       setFont (new Font ("Arial", Font.PLAIN, 14));
13       add(t); add(boxNegrito); add(boxItalico);
14       boxNegrito.addItemListener(new Controlador());
15       boxItalico.addItemListener(new Controlador());
16       setSize (250,100);
17       setVisible(true);
18    }
19    private class Controlador implements ItemListener {
20       public void itemStateChanged(ItemEvent e) {
21         if (e.getSource() == boxNegrito)
22           if (e.getStateChange() == ItemEvent.SELECTED)
23                negrito = Font.BOLD;
24           else negrito = Font.PLAIN;
25         if (e.getSource() == boxItalico)
26           if (e.getStateChange() == ItemEvent.SELECTED)
27                italico = Font.ITALIC;
28           else italico = Font.PLAIN;
29           t.setFont(new Font("Arial",negrito+italico,14));
30           t.repaint();
31         }
32    }
33 }
```

A classe `UmaOpção2` do quarto painel, colocado no lado direito na segunda linha da janela, segue o mesmo padrão de programação:

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  class UmaOpção2 extends JPanel {
4      private JRadioButton boxNegrito =
5                   new JRadioButton("Negrito");
6      private JRadioButton boxItalico =
7                   new JRadioButton("Itálico");
8      private ButtonGroup  grupo = new ButtonGroup();
9      private JTextField t  =
10        new JTextField("  Teste do Estilo",10);
11     private int estilo = Font.PLAIN;
12     public UmaOpção2 () {
13         setLayout(new FlowLayout());
14         t.setFont (new Font("Arial",estilo, 14));
15         add(t); add(boxNegrito); add(boxItalico);
16         grupo.add(boxNegrito);
17         grupo.add(boxItalico);
18         boxNegrito.addItemListener(new Controlador());
19         boxItalico.addItemListener(new Controlador());
20         setSize (250,100);
21         setVisible(true);
22     }
23     private class Controlador implements ItemListener {
24         public void itemStateChanged(ItemEvent e) {
25            if (e.getSource() == boxNegrito)
26                estilo = Font.BOLD;
27            if (e.getSource() == boxItalico)
28                estilo = Font.ITALIC;
29            t.setFont(new Font("Arial", estilo, 14));
30            t.repaint();
31         }
32     }
33 }
```

O quinto painel, localizado no lado esquerdo na terceira linha, é definido pela classe `Vinhos2`:

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import javax.swing.*;
4  class Vinhos2 extends JPanel {
5     private String uvas[]  = {
6       "merlot", "cabernet",  "sirah",
7       "carménère", "chardonnay" };
8     private JComboBox<String> caixa =
9                       new JComboBox<String>(uvas);
10    private JLabel escolha = new JLabel(uvas[0]);
11    private JLabel rotulo  =
12                      new JLabel("Vinho escolhido:");
13
14    public Vinhos2() {
15       setLayout(new FlowLayout());
16       caixa.addItemListener(new Ouvinte());
17       caixa.setMaximumRowCount(3);
18       add(caixa);
19       add(rotulo);
20       add(escolha);
21       setSize(340,150);
22       setVisible(true);
23    }
24    private class Ouvinte implements ItemListener {
25       public void itemStateChanged( ItemEvent event ) {
26         if (event.getStateChange() == ItemEvent.SELECTED)
27            escolha.setText(uvas[caixa.getSelectedIndex()]);
28       }
29    }
30 }
```

E o sexto painel é o do sistema de compras implementado pela classe `Compras2` e está colocado no lado direito da terceira linha da Fig. 20.1.

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  import java.util.List;
4  class Compras2 extends JPanel {
5    private final String nomes[] = {
6        "Lusíadas", "Don Quixote", "Relatividade",
7        "Linguagem Java", "Big Bang", "Grande Sertão",
8        "Steve Jobs", "Sapiens", "Gabriela",
9        "Os Sertões", "Dom Casmurro" };
10   private JList<String> livros = new JList<String>(nomes);
11   private JButton comando = new JButton(">>>");
12   private JList<String> carrinho = new JList<String>();
13   class Ouvinte implements ActionListener {
14     public void actionPerformed(ActionEvent e) {
15       List<String> selecao = livros.getSelectedValuesList();
16       Object[] a = selecao.toArray();
17       String[] s = new String[s1.length];
18       for (int i=0; i<s1.length ; i++) s[i] = (String)a[i];
19       carrinho.setListData(s);
20     }
21   }
22   public Compras2() {
23     setLayout(new FlowLayout());
24     livros.setVisibleRowCount(5);
25     livros.setSelectionMode(
26          ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
27     add(new JScrollPane(livros));
28     comando.addActionListener(new Ouvinte());
29     add(comando);
30     carrinho.setFixedCellWidth(80);
31     carrinho.setVisibleRowCount(3);
32     add(new JScrollPane(carrinho));
33     setSize(400,120);
34     setVisible(true);
35   }
36 }
```

20.2 Remoção e inclusão de Componentes

Componentes podem dinamicamente ser adicionados ou removidos da camada de conteúdo de painéis.

O programa apresentado a seguir mostra como incluir e remover componentes de painéis. Inicialmente, a janela da Fig. 20.2 é gerada, na qual as duas linhas superiores são botões de comando para executar a operação descrita em seus rótulos. E as duas linhas inferiores são componentes a ser removidos ou re-inseridos.

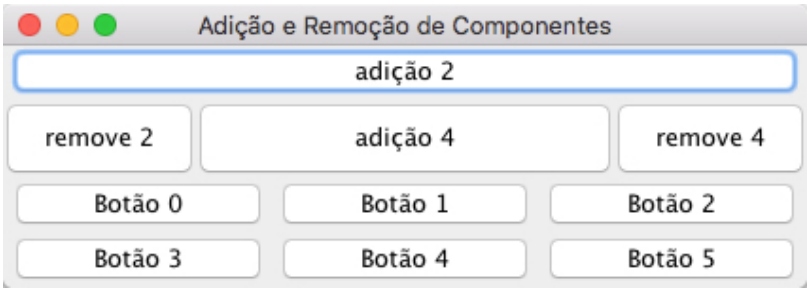


Figura 20.2 Adição e Remoção: Tela Inicial

As Fig. 20.3, 20.4 e 20.5 mostram o efeito de cliques, nessa ordem, nos botões comandos `remove 4`, `remove 2` e `adição 4`.

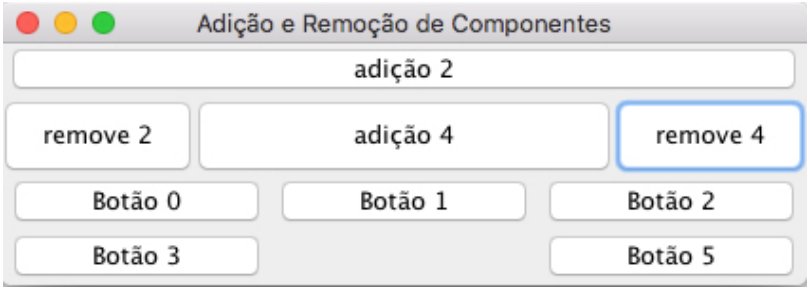


Figura 20.3 Adição e Remoção: Após clique em `remove 4`

Para a implementação desse programa exemplo, as operações `add` e `remove` da classe `JPanel`, herdadas de `Container`, foram usadas. É interessante observar que os componentes removidos foram re-inseridos exatamente no mesmo mesmo lugar em que es-

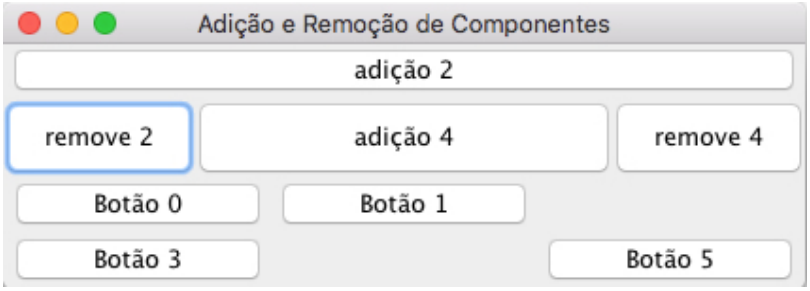


Figura 20.4 Adição e Remoção: Após clique em `remove 2`

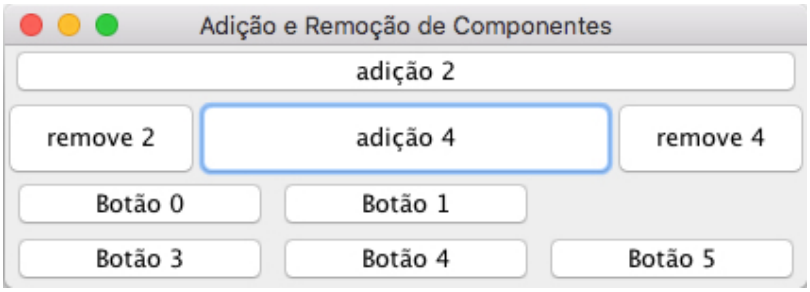


Figura 20.5 Adição e Remoção: Após clique em `adicao 4`

tavam no painel. E as operações de `JPanel` relacionadas à inclusão e remoção de componentes em painéis são as seguintes:

- `Component add(Component c)`:
  adiciona o componente `c` no fim do contêiner.
- `Component add(Component c, int n)`:
  adiciona o componente `c` na posição `n` do contêiner.
- `void remove(int n)`:
  remove o `n`-ésimo componente do contêiner.
- `void remove(Component c)`:
  remove o componente `c` do contêiner.
- `void removeAll()`:
  remove todos os componentes do contêiner.

Cada componente incluído em um painel tem uma referência distinta. Isso significa que a re-inclusão de um mesmo componente substitui sua inclusão anterior.

A classe **UsaOrganizaComponentes** define o programa principal padrão usado em todos os exemplos.

```
1 import java.awt.*; import java.awt.event.*;
2 import javax.swing.*;
3 class UsaOrganizaComponentes {
4   public static void main (String args[]) {
5     JFrame j = new OrganizaComponentes();
6     j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7    }
8 }
```

A classe **OrganizaComponentes**, usado no programa principal, é uma extensão de **JFrame**, que cria um painel com botões desejados e também exerce o papel de classe ouvinte de seus eventos de ação.

```
1 import java.awt.*; import java.awt.event.*;
2 import javax.swing.*;
3 public class OrganizaComponentes extends JFrame
4                                  implements ActionListener {
5     private JPanel  panel = new JPanel();
6     private JButton botoes[] = new JButton[6];
7     private JButton remocao2  = new JButton("remove 2");
8     private JButton remocao4  = new JButton("remove 4");
9     private JButton adicao2   = new JButton("adição 2");
10    private JButton adicao4   = new JButton("adição 4");
11
12    public OrganizaComponentes(){...}
13
14    public void actionPerformed(ActionEvent e) {...}
15 }
```

E a sua construtora tem a seguinte implementação, na qual destaca-se a tentativa de inserir cada botão de comando, nas linhas 8 e 9, duas vezes, que não tem efeito, pois cada botão é inserido somente um vez.

Observe também o uso do objeto **this**, no trecho da linha de 21 a 24, como o ouvinte de cliques em botões.

```
1  public OrganizaComponentes() {
2     super ("Adição e Remoção de Componentes");
3
4     // Prepara o painel de botões
5     buttonPanel.setLayout(new GridLayout (2,3));
6     for (int i = 0; i < botoess.length; i++) {
7       botoes[i] = new JButton ("Botão " + i);
8       panel.add(botoes[i]);
9       panel.add(botoes[i]); //somente um inserido
10    }
11
12    //Adiciona Componentes na Janela
13    Container c = getContentPane();
14    c.add(remocao2,BorderLayout.WEST);
15    c.add(remocao4,BorderLayout.EAST);
16    c.add(adicao2,BorderLayout.NORTH);
17    c.add(adicao4,BorderLayout.CENTER);
18    c.add(panel, BorderLayout.SOUTH);
19
20    //Associa Ouvinte aos Botões
21    remocao2.addActionListener(this);
22    remocao4.addActionListener(this);
23    adicao2.addActionListener(this);
24    adicao4.addActionListener(this);
25    setSize (425,150);
26    setVisible(true);
27 }
```

E o tratador de eventos de clique de *mouse*, implementado a seguir, simplesmente identifica o originador do evento e dá sequência à operação de remoção ou adição do componente identificado.

A operação **repaint** deve ser ativada no fim do tratador para a atualização do painel após remoções e adições.

```
1  public void actionPerformed(ActionEvent e) {
2     if  (e.getSource()== remocao2)
3          panel.remove(botoes[2]);
4     else if (e.getSource()== remocao4)
5               panel.remove(botoes[4]);
6     else if (e.getSource()== adicao2)
7               panel.add(botoes[2]);
8     else if (e.getSource()== adicao4)
9               panel.add(botoes[4]);
10    panel.repaint();
11 }
```

20.3 Conclusão

O posicionamento de componentes gráficos em uma janela pode tornar-se uma tarefa complexa, pois a inserção de cada novo componente pode afetar o posicionamento dos demais.

O componente `JPanel` permite melhor estruturar o leiaute dos componentes de forma hierárquica, minimizando o impacto de cada inserção.

Exercícios

1. Quais são as diferenças entre um objeto `JPanel` e um `Frame`, no sentido que que onde cada um pode substituir o outro?
2. Para que serve a camada *Glass Pane* de uma janela?
3. No programa da subseção 20.2, o painel e os seus componentes foram declarados como atributos da seguinte classe:

```
public class OrganizaComponentes extends JFrame
                              implements ActionListener{
  private JPanel  panel = new JPanel();
  private JButton botoes[] = new JButton[6];
  private JButton remocao2  = new JButton("remove 2");
```

```
    private JButton remocao4  = new JButton("remove 4");
    private JButton adicao2   = new JButton("adicao 2");
    private JButton adicao4   = new JButton("adicao 4");
    public OrganizaComponentes(){...}
    public void actionPerformed(ActionEvent e) {...}
}
```

pergunta-se se essas declarações desses componentes poderiam ser movidas para dentro da função construtora dessa classe?

Notas bibliográficas

Um dos melhores texto para estudar e aprofundar programação de interfaces gráfica é o livro *Java - Como programar* dos Deitels[10]. Os capítulos 11, 12 e 22 deste livro apresentam muitos exemplos ilustrativos de uso dos pacotes `Swing` e `AWT`. Vale a pena ler.

Capítulo 21

GUI: Mouse e Teclado

Teclado e o *mouse* são a principal ferramenta de interação do usuário com o programa. Esses dois dispositivos geram diversos eventos que precisam ser devidamente tratados para refletir as ações de seu usuário. Clique e movimentação de *mouse*, pressionamento e liberação de teclas são os eventos mais comuns.

A todos esses eventos devem ser associados tratadores e a todos os componentes gráficos com os quais o usuário interagem precisam de ouvintes apropriados.

Inicialmente, eventos de *mouse* são apresentados, depois, os de teclado e, por fim, o problema de difusão de eventos para muitos ouvintes é discutido e ilustrado.

21.1 Eventos de mouse

Os eventos gerados pelo *mouse* podem ser disparados sobre qualquer componente gráfico pelas ações do mouse como clicar e arrastar, e seus descritores devem informar aos seus tratadores as coordenadas `(x,y)` da tela onde os eventos tiveram origem.

Esses eventos são capturados por objetos ouvintes cujos tipos são de uma das interfaces `MouseListener`, `MouseMotionListener` e `MouseWheelListener`, cada uma com a função de capturar eventos específicos, a saber:

- `MouseListener`:
  captura eventos como cliques de *mouse* e mudança de posição do ponteiro do *mouse*.
- `MouseMotionListener`:
  captura eventos que ocorrem quando o botão do *mouse* é pressionado e o ponteiro é movido simultaneamente.
- `MouseWheelListener`;
  captura eventos gerados pela rotação da rodinha do *mouse*.

Os métodos de `MouseListener` e `MouseMotionListener` são chamados quando têm-se as três condições: o *mouse* gera o evento, o *mouse* interage com um componente e objetos ouvintes apropriados foram registrados no componente. Os métodos dessas interfaces estão resumidos na seguinte tabela:

Interface	Métodos Tratadores de Eventos	Parâmetro/ Acessadores	Origem
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent getClickCount getX, getY getPoint getButton getModifiers getMouseModifierText isPopupTrigger	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent ...	Component

O descritor de eventos gerados pelo *mouse* é um objeto da classe `MouseEvent`, montado pelo originador do evento e passado como parâmetro ao correspondente tratador, e pode ser manipulado pelas seguintes operações:

- `int getClickCount()`:
  retorna número de cliques rápidos e consecutivos.
- `int getX()`, `int getY()` e `Point getPoint()`:
  retornam coordenadas (x,y) relativas ao componente do mouse onde evento ocorreu.
- `int getButton()`:
  retorna qual botão do mouse mudou de estado. Os botões são dados pelas constantes: `NOBUTTON, BUTTON1, BUTTON2` e `BUTTON3`.
- `int getModifier()`:
  retorna um inteiro `m` descrevendo as teclas modificadoras, e.g., (*Shift, Ctl+Shift*) ativas durante evento.
- `String getMouseModifierText(int m)`:
  retorna uma cadeia de caracteres descrevendo as teclas modificadoras contidas no modificador retornado por `getModifier`.
- `boolean isPopUpTrigger()`:
  retorna `true` se o evento de mouse deve causar o aparecimento de um menu pop-up.

Os eventos encaminhados a ouvintes do tipo `MouseListener` são tratados por um dos seguintes métodos:

- `void mousePressed(MouseEvent e)`:
  botão do mouse pressionado com cursor sobre um componente.
- `void mouseClicked(MouseEvent e)`:
  botão do mouse pressionado e liberado sobre um componente sem movimentação do cursor (clique).
- `void mouseReleased(MouseEvent e)`:
  botão do mouse liberado depois de ser pressionado. Sempre precedido por `mousePressed`.
- `void mouseEntered(MouseEvent e)`:
  chamado quando cursor do mouse entra nos limites de um componente.

- `void mouseExited(MouseEvent e)`:
  chamado quando cursor do mouse deixa os limites de um componente.

Os tratadores de evento do tipo `MouseMotionListener` são os seguintes:

- `void mouseDragged(MouseEvent e)`:
  botão do mouse é pressionado e o cursor é movido. Sempre precedido por `mousePressed`.
- `void mouseMoved (MouseEvent e)`:
  mouse movido com o cursor sobre um componente.

E o único tratador de evento do tipo `MouseWeelListener` é:

- `void mouseWheelMoved(MouseWheelEvent e)`:
  invocado quando a rodinha do *mouse* gira.

onde o descritor de eventos `MouseWheelEvent` tem as seguintes operações:

- `void getPreciseWheelRotation()`:
  retorna o número de *cliques* que a rodinha movimentou-se.
- `int getScrollAmount()`:
  retorna o número de unidades a ser deslocadas por *clique* da rodinha.
- `int getScrollType()`:
  retorna o tipo de deslizamento associado ao evento.
- `int getUnitsToScroll()`:
  retorna a unidade usada no processo de movimentação da rodinha.
- `int getWheelRotation()`:
  retorna o número de *cliques* que a rodinha movimentou-se.
- `String paramString(()`:
  retorna a cadeia de caracteres que identifica o evento.

O programa `UsaEvtMouse`, apresentado a seguir, exibe inicialmente a janela da Fig. 21.1. E ao movimentar o *mouse* para dentro da janela tem-se a Fig. 21.2.

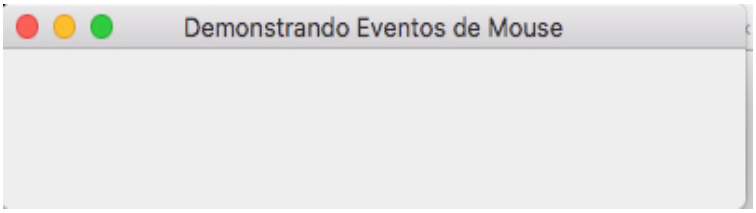


Figura 21.1 Movimentação do Mouse

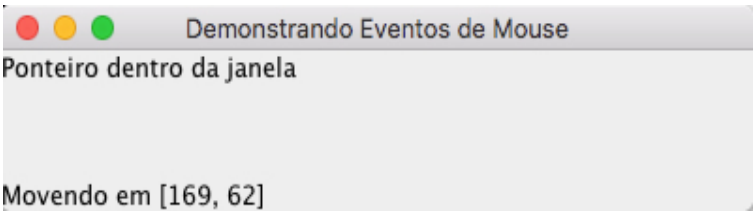


Figura 21.2 Movimentação do Mouse

A seguir, ao colocar o *mouse* fora da janela, tem-se a Fig. 21.3.

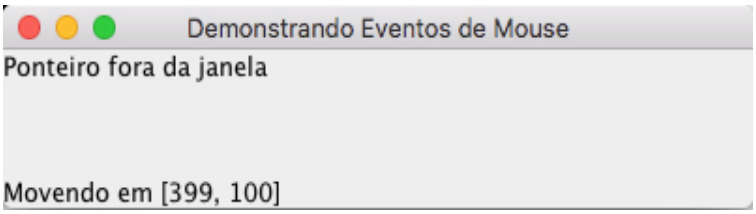


Figura 21.3 Movimentação do Mouse

Quando o *mouse* retorna para dentro da janela, tem-se a Fig. 21.4.

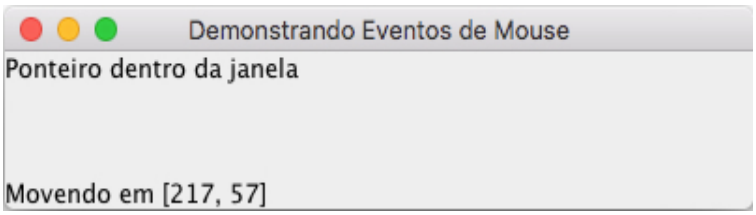


Figura 21.4 Movimentação do Mouse

O clique do *mouse* com o ponteiro na posição `(180,59)` gera a Fig. 21.5. E se nesse clique, sem liberar o botão do *mouse*, arrastá-lo para a posição `(26,46)`, e só então liberar o botão, tem-se a Fig. 21.6.

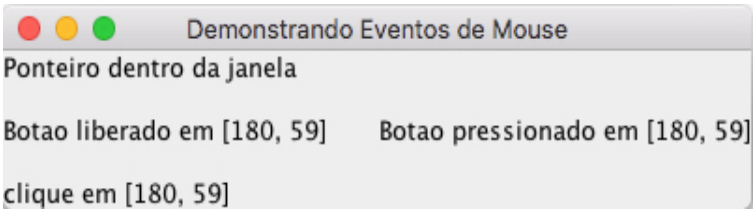


Figura 21.5 Movimentação do Mouse

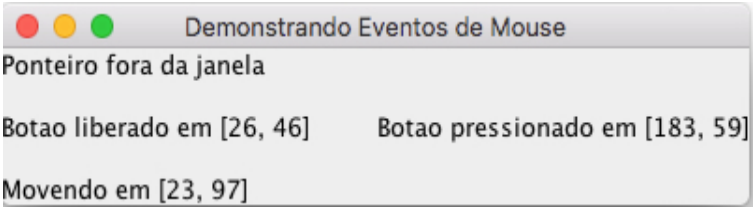


Figura 21.6 Movimentação do Mouse

A classe `EvtMouseDemo` é a responsável pela construção da janela exibida nas figuras acima e também é ouvinte de eventos dos tipos `MouseListener` e `MouseMotionListener`.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class EvtMouseDemo extends JFrame
5            implements MouseListener, MouseMotionListener {
6    private JLabel status1, status2, status3, status4;
7    private Container c = getContentPane();
8    public EvtMouseDemo() {...}
9    "Operações de tratamento de evento";
10 }
```

A construtora de `EvtMouseDemo` cria quatro rótulos para relatar as ações do *mouse* e instala no componente correspondente à janela principal os ouvintes de seus cliques e movimentos.

```
1  public EvtMouseDemo() {
2     super ("Demonstrando Eventos de Mouse");
3     status1 = new JLabel();
4     status2 = new JLabel();
5     status3 = new JLabel();
6     status4 = new JLabel();
7     c.add(status1, BorderLayout.SOUTH);
8     c.add(status2, BorderLayout.NORTH);
9     c.add(status3, BorderLayout.EAST);
10    c.add(status4, BorderLayout.WEST);
11    addMouseListener (this);
12    addMouseMotionListener (this);
13    setSize(400,100);
14    setVisible(true);
15   }
```

Os dois tratadores de eventos de `MouseMotionListener` têm as implementações apresentadas a seguir, que repetidamente recuperam do descritor de eventos a elas passado as coordenadas do *mouse* na medida em que ele se movimenta, e as exibem nos rótulos apropriados.

```
1  public void mouseDragged (MouseEvent e) {
2     status1.setText ("Arrastando em [" +
3                 e.getX() + ", " + e.getY() + "]");
4  }
5  public void mouseMoved (MouseEvent e) {
6     status1.setText ("Movendo em [" +
7                 e.getX() + ", " + e.getY() + "]");
8  }
```

Os tratadores de eventos de `MouseListener`, cujas implementações são mostradas a seguir, também repetidamente recuperam

do descritor de evento a eles passados as coordenadas do *mouse* do ponto em que os eventos ocorreram e as exibem nos rótulos apropriados.

```
1  public void mouseClicked (MouseEvent e) {
2     status1.setText("clique em [" + e.getX() + ", " +
3                                        e.getY() + "]");
4  }
5  public void mousePressed (MouseEvent e) {
6     status3.setText("Botao pressionado em [" + e.getX() +
7                                      ", " + e.getY()+ "]");
8  }
9  public void mouseReleased (MouseEvent e) {
10    status4.setText("Botao liberado em [" + e.getX() + ", "
11                                   + e.getY() + "]");
12 }
13 public void mouseEntered (MouseEvent e) {
14    status2.setText ("Ponteiro dentro da janela");
15 }
16 public void mouseExited (MouseEvent e) {
17    status2.setText ("Ponteiro fora da janela");
18 }
```

Finalmente, o programa principal padrão:

```
1  public class UsaEvtMouse {
2      public static void main (String args[]){
3         JFrame j = new EvtMouseDemo ();
4         j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5      }
6  }
```

O próximo exemplo mostra como programar o tratador de evento de arrastamento de *mouse* e o método `paint` de `JFrame` para usar o *mouse* como se fosse uma caneta.

A ideia é criar uma janela em branco e na medida em que o *mouse* for movimentado na tela com o botão pressionado, suas co-

Figura 21.7 Tela para Desenho

ordenadas são recuperadas pelo tratador de eventos e nessa posição desenha-se um pequeno círculo com a função **paint** de **JFrame**, ativada por meio do método **repaint**, conforme mostra Fig. 21.7.

```
1  public class Caneta extends JFrame {
2      private int x = -10; y = -10;
3      public Caneta() {
4         super("Uma tela eletrônica");
5         getContentPane().add(
6              new Label ("Arraste o mouse para desenhar"),
7                                        BorderLayout.SOUTH);
8
9         addMouseMotionListener( ... );
10
11        setSize(300,150);
12        setVisible(true);
13     }
14     public void paint(Graphics g){g.fillOval(x,y,4,4);}
15     public static void main(String args[]){
16        JFrame j = new Caneta ();
17        j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     }
19 }
```

Para a implementação do tratador de evento, usou-se a classe adaptadora **MouseMotionAdapter** para não ter que definir o outro tratador, o **mouseMoved**, sem função no presente exemplo. O

tratador implementado, que apenas atualiza as coordenadas de desenho `(x,y)` a partir do descritor do evento que lhe foi passado e força um `repaint`, é associado à janela `Caneta` por:

```
addMouseMotionListener (
      new MouseMotionAdapter() {
            public void mouseDragged (MouseEvent e) {
                  x = e.getX();
                  y = e.getY();
                  repaint();
            }
       }
);
```

21.2 Eventos de teclado

Eventos de teclado são gerados quando teclas são pressionadas ou liberadas e podem ser tratados por ouvintes do tipo `KeyListener`, que devem estar associados ao componente que tem o foco do teclado no momento em que o usuário aciona as teclas. Ouvintes do tipo `KeyListener` têm três métodos para tratamento de eventos:

- `keyPressed(KeyEvent e)`:
  chamado em resposta ao pressionamento de qualquer tecla.

- `keyTyped(KeyEVent e)`:
  chamado em resposta ao pressionamento de tecla que não é uma tecla de ação (*Home, End, Page Down, Page Up, Num, Caps Lock*, etc).

- `keyReleased(KeyEvent e)`:
  chamado quando uma tecla é liberada, ocorre depois de um evento `keyPressed` ou `keyTyped` ter ocorrido.

A classe descritora de eventos `KeyEvent` define, entre outros recursos, uma série de constantes simbólicas que denotam valores relativos ao teclado, como códigos virtuais para as teclas, todos escritos com o prefixo `VK_`. Por exemplo, `VK_A`, `VK_B`, `VK_SHIFT`, que denotam, respectivamente, a tecla `A`, `B` e *Shift*.

Os códigos dos caracteres digitados podem ser recuperados pelos métodos `getKeyCode()`, que informa o código virtual da tecla, e `getKeyChar()`, que informa o caractere digitado, de `KeyEvent`, cujos principais métodos são os seguintes:

- `char getKeyChar()`:
  devolve o caractere digitado.
- `int getKeyCode()`:
  devolve o código virtual do caractere digitado.
- `static String getKeyText(int codTecla)`:
  devolve texto descrevendo o código virtual da tecla.
  Ex: `getKeyText(KeyEvent.VK_END)` devolve *End*.
- `String getKeyModifiersText(int modificadores)`:
  devolve a descrição das teclas modificadoras como *Shift, Ctrl, Ctrl+Shift*. Parâmetro é obtido pelo método `getModifiers()`.
- `boolean isAltDown()`:
  idem para tecla *Alt*.
- `boolean isControlDown()`:
  idem para tecla *Ctrl*.
- `boolean isShiftDown()`:
  idem para tecla *Shift*.

Vários eventos são gerados quando uma tecla é digitada, por exemplo:

- Ao digitar uma letra `"A"`, pressionando *Shift* e a tecla A, ocorrem os seguintes eventos e ações:

1. pressionada a tecla *Shift*:
   chamado `keyPressed` com `VK_SHIFT`;
2. pressionada a tecla A:
   chamado `keyPressed` com `VK_A`;
3. digitada `"A"`, i.e., a tecla chegou ao fundo:
   chamado `keyTyped` com `"A"`;
4. liberada a tecla A:
   chamado `keyReleased` com `VK_A`;
5. liberada a tecla SHIFT:
   chamado `keyReleased` com `VK_SHIFT`.

- Ao digitar uma letra `"a"`, pressionando a tecla A, ocorrem os seguintes eventos e ações:

1. pressionada a tecla A:
   chamado `keyPressed` com `VK_A`;
2. digitada `"a"`, i.e., tecla chegou ao fundo:
   chamado `keyTyped` com `"a"`;
3. liberada a tecla A:
   chamado `keyReleased` com `VK_A`.

A tabela a seguir resume os três métodos de `KeyListener`, todos com um único parâmetro do tipo `KeyEvent`.

Interface	Método	Parâmetro/ Acessadores	Origem
KeyListener	keyPressed keyReleased keyTyped	KeyEvent getKeyChar getKeyCode getKeyModifiersText getKeyText isActionKey	Component

O originador de eventos tratados por ouvintes `KeyListener` pode ser qualquer componente que esteja no foco da interação.

O programa `UsaTeclado`, apresentado a seguir, exibe na tela as ações de uso do teclado pelo usuário para destacar o que acontece internamente quando teclas são pressionadas ou liberadas.

Fig. 21.8 mostra em cada área de texto o efeito de: (1) digitar a tecla A, (2) pressionar *Shift* ao mesmo tempo que digitar A, (3) digitar a tecla 1, (4) pressionar *Ctrl*, (5) digitar tecla *Page Down* e (6) pressionar simultaneamente as teclas *Command* e *Shift*.

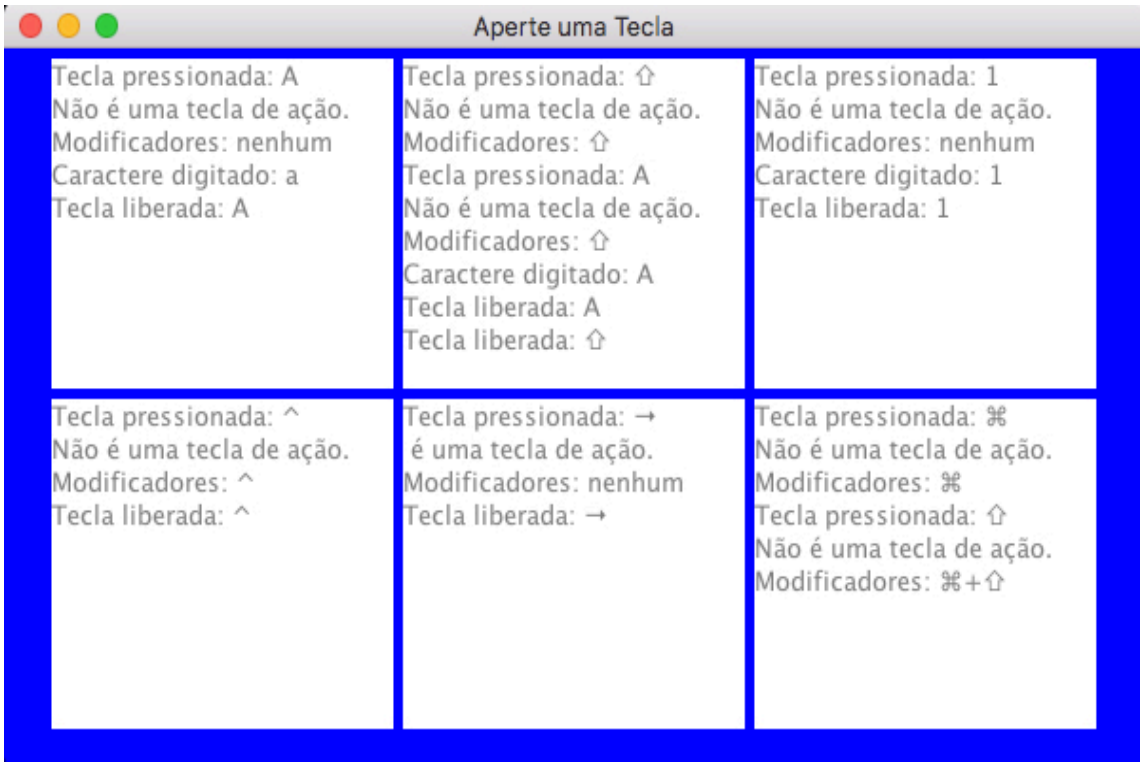


Figura 21.8 Eventos de Teclado

Quando uma tecla é acionada, três eventos relativos à interface `KeyListener` são gerados: `keypressed`, `keyTyped` e `keyReleased`. As ocorrências desses eventos estão identificadas na Fig. 21.8 como `Tecla pressionada`, `Caractere digitado` e `Tecla liberada`.

A classe `UsaTeclado` monta e exibe a janela da Fig. 21.8 e implementa os tratadores de eventos do tipo `KeyListener`. A

janela consiste em seis áreas de texto do tipo `JTextArea`.

```
1  import javax.swing.*; import java.awt.*;
2  import java.awt.event.*;
3  public class UsaTeclado extends JFrame
4                                      implements KeyListener {
5    private int MAX = 6, indice = 0, contadorDePressed = 0;
6    private String impresso = "";
7    private JTextArea[] texto = new JTextArea[MAX];
8    public UsaTeclado() { ... }
9    public void keyPressed (KeyEvent e) { ... }
10   public void keyTyped (KeyEvent e) { ... }
11   public void keyReleased (KeyEvent e) { ... }
12   public static void main (String args[]) {
13      JFrame j = new UsaTeclado();
14      j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15   }
16 }
```

A construtora de `UsaTeclado` configura a janela da aplicação, adicionando-lhe seis áreas de texto e a ela vinculando o objeto ouvinte para tratamento dos eventos de teclado ocorridos.

```
1  public UsaTeclado() {
2     super ("Aperte uma Tecla");
3     Container c = getContentPane();
4     c.setLayout(new FlowLayout());
5     c.setBackground(Color.yellow);
6     for (int i = 0; i < MAX; i++) {
7         texto[i]= new JTextArea(10,15);
8         texto[i].setEnabled(false);
9         c.add(texto[i]);
10    }
11    addKeyListener(this);
12    setSize(600,370); setVisible(true);
13 }
```

Toda vez que, estando a janela da aplicação em foco, uma tecla for pressionada, o seguinte método **keypressed** é acionado para identificar a tecla acionada e preencher uma das áreas de texto.

```
1  public void keyPressed (KeyEvent e) {
2    String temp = KeyEvent.getKeyModifiersText(
3                                              e.getModifiers());
4    String line = "Tecla pressionada: " +
5              KeyEvent.getKeyText(e.getKeyCode()) + "\n" +
6              (e.isActionKey() ? "": "Não") +
7               " é uma tecla de ação." + "\n" +
8                "Modificadores: " +
9                 (temp.equals("") ? "nenhum" : temp) + "\n";
10   impresso = impresso + line;
11   texto[indice % MAX].setText(impresso);
12   contadorDePressed++;
13 }
```

Após o acionamento do tratador **keyPressed**, ocorre a chamada ao tratador **keyTyped**, descrito a seguir, que informa na área de texto corrente qual foi o caractere digitado.

```
1  public void keyTyped (KeyEvent e) {
2    String line = "Caractere digitado: " +
3                                     e.getKeyChar() + "\n";
4    impresso = impresso + line;
5    texto[indice].setText(impresso);
6  }
```

A variável **contadorDePressed**, declarada em **UsaTeclado** e usada em **keyPressed** e **keyReleased**, é o sinalizador de mudança de área de texto corrente usada para exibir os efeitos do uso do teclado. Inicialmente tem o valor `0`, e sempre que uma tecla é pressionada, essa variável é incrementada, e quando a mesma tecla é liberada, seu valor é decrementado, de forma que quando seu valor voltar a ser `0`, uma nova área de texto deve ser usada.

```
1  public void keyReleased (KeyEvent e) {
2    String line = "Tecla liberada: " +
3             KeyEvent.getKeyText(e.getKeyCode()) + "\n";
4    impresso = impresso + line;
5    texto[indice].setText(impresso);
6    contadorDePressed--;
7    if (contadorDePressed == 0) {
8           impresso = "";
9           indice = (indice + 1) % MAX;
10   }
11 }
```

21.3 Multidifusão de eventos

Objetos originadores de eventos da família AWT suportam um modelo de multidifusão para os ouvintes neles registrados. Isso significa que um mesmo evento que seja do interesse em muitas partes do programa pode ser enviado para mais de um ouvinte, para que cada um possa produzir o efeito associado.

Figura 21.9 Difusão de Ouvintes

O programa `MulticastTest` a seguir ilustra o processo de difusão de eventos usando uma janela com dois botões: `Cria Janela`,

para comandar a criação de uma nova janela, e um outro, o `Fecha Todas`, para fechar todas as janelas que foram criadas, como mostra a Fig. 21.9.

A Fig. 21.10 mostra a tela do computador depois de quatro cliques no botão `Cria Janela`.

Figura 21.10 Difusão após vários `Cria Janela`

O tratador de eventos associados ao botão `Cria Janela` deve criar a cada clique uma das janelas mostradas na Fig.21.10, e ao botão `Fecha Todas` devem ser associados ouvintes para destruir cada uma das janelas criadas.

Na solução que se segue, sempre que uma janela for criada, um ouvinte destruidor dessa janela será associado ao botão `Fecha Todas` para que, quando houver um clique nesse botão, todos os ouvintes associados possam cumprir seu papel.

A janela criada na classe `MultiDifusão` cria, após as pro-

vidências de praxe, um painel de difusão na linha 11.

```
1  class MultiDifusão extends JFrame {
2     public MultiDifusão() {
3       setTitle("Difusão de Eventos");
4       setSize(300,150);  setVisible(true);
5       addWindowListener(new WindowAdapter(){
6          public void windowClosing(WindowEvent e){
7             System.exit(0);
8          }
9       });
10      Container c = getContentPane();
11      c.add(new PainelDeDifusão());
12    }
13 }
```

A classe `PainelDeDifusão`, apresentada a seguir, implementa o painel adicionado à janela acima.

A variável estática `contador`, declarada na linha 5, identifica cada janela criada. E o ouvinte do botão `Cria Janela` é o próprio painel, que implementa o tratador `actionPerformed`.

```
1  import java.awt.*; import java.awt.event.*;
2  import javax.swing.*;
3  class PainelDeDifusão extends JPanel
4                            implements ActionListener {
5     private static int contador = 1;
6     private JButton novo  = new JButton("Cria Janela");
7     private JButton apaga = new JButton("Fecha Todas");
8     public PainelDeDifusão(){
9        add(novo); novo.addActionListener(this);
10       add(apaga);
11    }
12   public void actionPerformed(ActionEvent evt){ ... }
13 }
```

Os seguintes métodos de `JFrame` são usados nas implementações de `actionPerformed` de `PainelDeDifusão` e de `SimpleFrame` apresentadas abaixo:

- `dispose ()`:
  fecha janela e recupera os recursos do sistema utilizados, liberando a área de memória ocupada pela objeto `JFrame`.
- `setLocation(int x, int y)`:
  posiciona a janela nas coordenadas `(x,y)` da tela.

O método `actionPerformed` de `PainelDeDifusão` é acionado em resposta a cliques no botão `Cria Janela`. Ele cria uma nova janela do tipo `SimpleFrame` e a associa ao botão `Fecha Todas` como um de seus ouvintes. Objetos do tipo `SimpleFrame` é um ouvinte `ActionListener`, que, quando acionado, simplesmente elimina-se do programa via a operação `dispose`, como indicado na linha 14 do código abaixo.

```
1  public void actionPerformed(ActionEvent evt){
2     SimpleFrame f = new SimpleFrame();
3     int c = PainelDeDifusão.countador++;
4     f.setTitle("Janela " + c);
5     f.setSize(200,200);
6     f.setLocation(300 + 30 * c, 60 * c);
7     f.setVisible(true);
8     apaga.addActionListener(f);
9  }
10
11 class SimpleFrame extends JFrame implements ActionListener{
12    public void actionPerformed(ActionEvent evt){
13       PainelDeDifusão.contador = 1;
14       dispose();
15    }
16 }
```

O programa `UsaMultiDifusão` inicia-se criando a janela que irá conter um painel com os dois botões descritos acima.

```
1 public class UsaMultiDifusão {
2   public static void main(String[] args) {
3     JFrame frame = new MultiDifusão();
4   }
5 }
```

21.4 Conclusão

Os recursos para programação de tratamento de eventos de *mouse* e teclado em interfaces gráficas em Java são extremamente poderosos com muitas dezenas de classes e centenas de métodos. Este capítulo apenas aborda pontos básicos dessa rica biblioteca, apenas para introduzir o iniciante no tema. Para aprofundar o assunto, recomendam-se os livros dos Deitels.

Exercícios

1. No programa da subseção 20.2, o painel e os seus componentes foram declarados como atributos de `OrganizaComponentes`, da seguinte forma:

```
public class OrganizaComponentes extends JFrame
                            implements ActionListener{
  private JPanel  panel     = new JPanel();
  private JButton botoes[]  = new JButton[6];
  private JButton remocao2  = new JButton("remove 2");
  private JButton remocao4  = new JButton("remove 4");
  private JButton adicao2   = new JButton("adicao 2");
  private JButton adicao4   = new JButton("adicao 4");
  public OrganizaComponentes(){...}
  public void actionPerformed(ActionEvent e) {...}
}
```

pergunta-se se essas declarações poderiam ser movidas para dentro da função construtora dessa classe?

Notas bibliográficas

Um dos melhores texto para estudar e aprofundar programação de interfaces gráfica é o livro *Java - Como programar* dos Deitels[10]. Os capítulos 11, 12 e 22 apresentam muitos exemplos ilustrativos de uso dos pacotes `swing` e `awt`. Vale a pena ler.

Capítulo 22

GUI: Componentes Avançados

Componentes do tipo `JTextField` somente têm espaço para uma linha de texto. Quando várias linhas são desejadas, `JTextArea` deve ser usado, com ou sem barras de rolagem. Muitas vezes, uma entrada de dados analógica, realizada por meio de uma barra deslizante da classe `JSlider`, é mais confortável de usar que digitar os valores numéricos desejados. Outros recursos gráficos muito importantes são os menus da família `JMenu` e a possibilidade de aninhar janelas via `JInternalFrame`.

22.1 Áreas de texto

A classe `JTextArea`, assim como `JTextField`, pertence à hierarquia de `JTextComponent` e serve para definir componentes que fornecem uma área de texto de manipulação de múltiplas linhas. Essa área, depois de preenchida, pode ser ou não editável pelo usuário. Áreas de texto têm externamente uma área de exibição cujo tamanho é definido no momento de criação do componente, quando especificam-se o número de linhas e o de colunas a ficar sempre visíveis. Internamente ao componente há uma área de armazenamento para texto de qualquer tamanho.

Dois métodos importantes de `JTextArea` são:

- `setEditable(boolean b)`: define se a caixa é editável ou

não. Se `b` for `false`, a caixa de texto passa a ser de leitura somente. Se `true`, se contéudo pode ser editado.

- `JTextArea(String t,int largura,int altura)`: constrói o componente com uma área visível de tamanho `largura` X `altura` e o inicializa com o texto `t`.

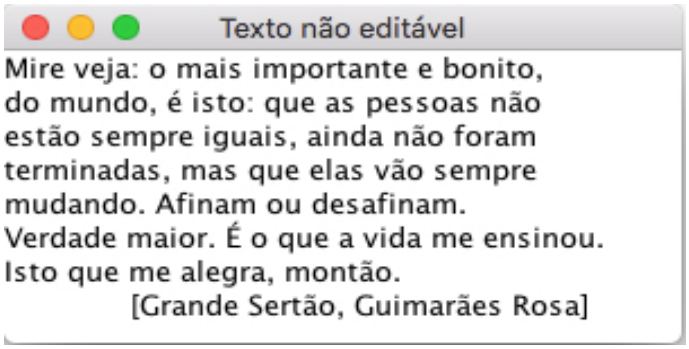


Figura 22.1 Área de Texto de Tamanho Fixo

A Fig. 22.1 exemplifica um componente `JTextArea` já preenchido com um texto pelo programa `UsaAreaDeTexto1`, que segue padrão aqui adotado para programa principal, criando um objeto da classe `AreaDeTexto1`, onde estão alojados os detalhes da montagem do componente e sua instalação na janela visível.

```
1 import java.awt.*;
2 import javax.swing.*;
3 import java.awt.event.*;
4 public class UsaAreaDeTexto1 {
5    public static void main (String args[]) {
6       JFrame j = new AreaDeTexto1();
7       j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8    }
9 }
```

A classe `AreaDeTexto1`, que é definida como sendo uma subclasse de `JFrame`, cria o componente `t` do tipo `JTextArea`, já

inicializado com a cadeia de caracteres `s` e o instala na camada de contéudo da janela.

```
1  import java.awt.*;
2  import javax.swing.*;
3  class AreaDeTexto1 extends JFrame {
4     private JTextArea t;
5     public AreaDeTexto1() {
6        super ("Texto não editável");
7        Container c = getContentPane();
8        String s =
9          "Mire veja: o mais importante e bonito,\n"+
10         "do mundo, é isto: que as pessoas não\n" +
11         "estão sempre iguais, ainda não foram\n" +
12         "terminadas, mas que elas vão sempre\n" +
13         "mudando. Afinam ou desafinam.\n" +
14         "Verdade maior. É o que a vida me ensinou.\n" +
15         "Isto que me alegra, montão.\n" +
16         "                [Grande Sertão, Guimarães Rosa]";
17       t = new JTextArea(s,10,40);
18       t.setEditable(false);
19       c.add(t);
20       setSize(300,160);
21       setVisible(true);
22    }
23 }
```

Nessa implementação, o tamanho 10 linhas X 40 colunas da parte visível do texto inserido na caixa de texto foi fixado no momento criação do componente. A parte do texto inserido que extrapolar os limites do visor da área ficará escondido. Se essa situação for uma real possibilidade, é recomendado instalar barras de rolagem na `JTextArea`, por meio de um objeto do tipo `JScrollPane`. As barras de rolagem são sempre duas: horizontal e vertical, para permitir deslocar o texto armazenado nas duas direções.

A classe `AreaDeTexto2` é uma cópia de `AreaDeTexto1`, exceto pela instalação da barras de rolagem na linha 18 do código abaixo.

```
1  import java.awt.*; import javax.swing.*;
2  class AreaDeTexto2 extends JFrame {
3     private JTextArea t;
4     public AreaDeTexto2() {
5        super ("Texto não editável");
6        Container c = getContentPane();
7        String s =
8          "Mire veja: o mais importante e bonito,\n"+
9          "do mundo, é isto: que as pessoas não\n" +
10         "estão sempre iguais, ainda não foram\n" +
11         "terminadas, mas que elas vão sempre\n" +
12         "mudando. Afinam ou desafinam.\n" +
13         "Verdade maior. É o que a vida me ensinou.\n" +
14         "Isto que me alegra, montão.\n" +
15         "                [Grande Sertão, Guimarães Rosa]";
16       t = new JTextArea(s,10,40);
17       t.setEditable(false);
18       c.add(new JScrollPane(t));
19       setSize(300,108); setVisible(true);
20    }
21 }
```

E o program principal segue o padrão que vem sendo usado.

```
1  import java.awt.*; import javax.swing.*;
2  import java.awt.event.*;
3  public class UsaAreaDeTexto2 {
4      public static void main (String args[]) {
5          JFrame j = new AreaDeTexto2();
6          j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7      }
8  }
```

Fig. 22.2 mostra uma área de texto com mais linhas que a área visível, e note que as barras estão devidamente instaladas.

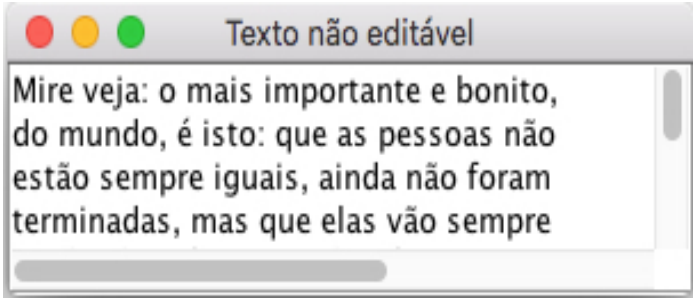


Figura 22.2 Visão inicial do Texto

E o deslocamento da barra vertical para baixo faz aparecer a parte final do texto.

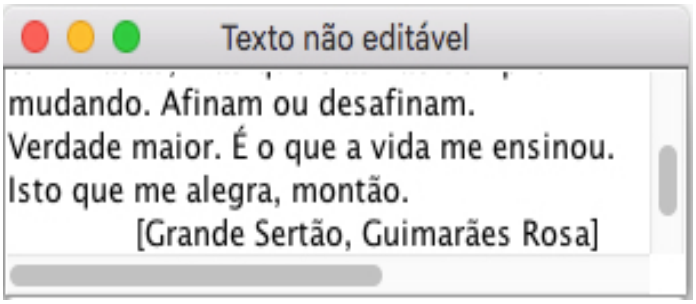


Figura 22.3 Area de Texto Deslocada

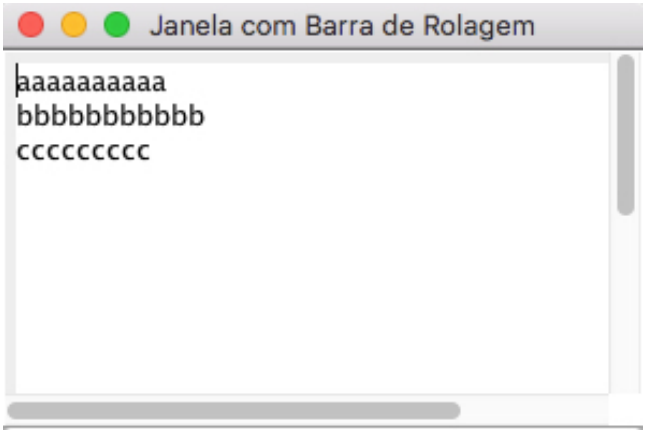


Figura 22.4 Barras de Rolagem em Janela

Barras de rolagem também podem ser usadas com componentes `JFrame`, como ilustra a Fig. 22.4, que é produzida pelo programa `UsaJanelaComScrollBar`, apresentado a seguir, e que instala as

barras de rolagem diretamente na camada de conteúdo do painel criado.

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  class JanelaComScrollBar extends JFrame {
6     public JanelaComScrollBar() {
7        setTitle("Janela com Barra de Rolagem");
8        setSize(300,200);
9        Container c = getContentPane();
10       JPanel j = new JPanel();
11       c.add(new JScrollPane(j));
12       String s ="aaaaaaaaaa\nbbbbbbbbbbb\nccccccccc";
13       JTextArea t = new JTextArea(s,20,30);
14       j.add(t);
15       setVisible(true);
16    }
17 }
18
19 public class UsaJanelaComScrollBar {
20    public static void main(String [ ] args) {
21       JFrame janela = new JanelaComScrollBar();
22       janela.addWindowListener(
23          new WindowAdapter(){
24             public void windowClosing(WindowEvent e){
25                System.exit(0);
26             }
27          }
28       );
29    }
30 }
```

22.2 Barras deslizantes

O componente `JSlider` também chamado de *controle deslizante* permite ao usuário selecionar valores deslocando com o ponteiro do *mouse* um indicador ao longo de uma barra que define um intervalo de valores inteiros. Um `JSlider` é como uma barra graduada com marcas de medidas, como exemplifica a Fig. 22.5

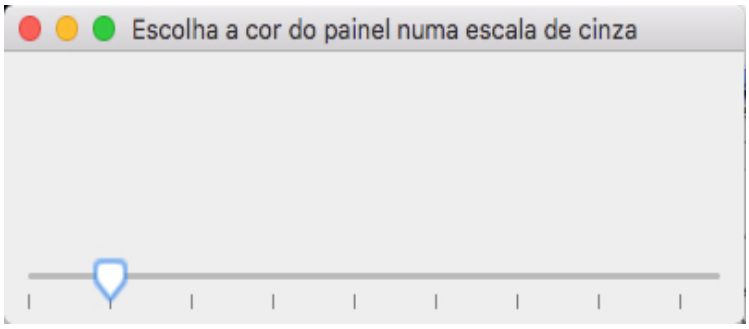


Figura 22.5 Janela com um JSlider

Componentes do tipo `JSlider` suportam eventos de mouse e de teclado. Uma barra deslizante pode ter orientação horizontal, definida pela constante inteira `SwingConstants.HORIZONTAL` ou vertical dada por `SwingConstants.VERTICAL`. Um `JSlider` horizontal tem seu valor mínimo posicionado no lado esquerdo, e o vertical, embaixo. Operações do componente `JSlider` incluem:

- `JSlider(int orient, int min, int max, int init)`: um dos construtores da classe, que define *orientação, valor mínimo, valor máximo* e *valor inicial* do componente.
- `void setMajorTickSpacing(int)`: determina o intervalo entre duas marcas visíveis do `JSlider`.
- `void setPaintTicks (boolean b)`: determina se as marcas de graduação são visíveis ou não.
- `void setPaintLabels(boolean b)`: determina se as marcas devem ou não ser rotuladas.

- `getValue()`:
  retorna o valor corrente indicado pela marca do componente `JSlider`.

O ouvinte de eventos de barras deslizantes é da hierarquia de `ChangeListener`, que tem as seguintes características:

Interface	Método	Parâmetro/ Acessadores	Origem
ChangeListener	stateChanged	ChangeEvent getSource	JSlider

onde `ChangeEvent` é classe do descritor do evento, cujo único método `getSource` permite identificar sua origem.

O programa `UsaSliderCores`, apresentado a seguir, permite exibir um painel na cor definida pelas intensidades de vermelho, verde e azul. A tabela de cores abaixo mostra as intensidades, valores do intervalo $[0, 255]$, que definem cores populares.

Constante Color	Cor	Valor RGB
`final static Color orange`	laranja	255,200,0
`final static Color cyan`	ciano	0,255,255
`final static Color magenta`	magenta	255,0,255
`final static Color yellow`	amarelo	255,255,0
`final static Color black`	preto	0,0,0
`final static Color white`	branco	255,255,255
`final static Color gray`	cinza	128,128,128
`final static Color lightGray`	cinza-claro	192,192,192
`final static Color darkGray`	cinza-escuro	64,64,64
`final static Color red`	vermelho	255,0,0
`final static Color green`	verde	0,255,0
`final static Color blue`	azul	0,0,255

O programa `UsaSliderCores` instala em uma janela, ao lado de um painel colorido, como na Fig. 22.6, três barras deslizantes: a vertical à esquerda, usada para definir a intensidade de vermelho, a horizontal, para a do verde, e a vertical à direita, para a do azul.

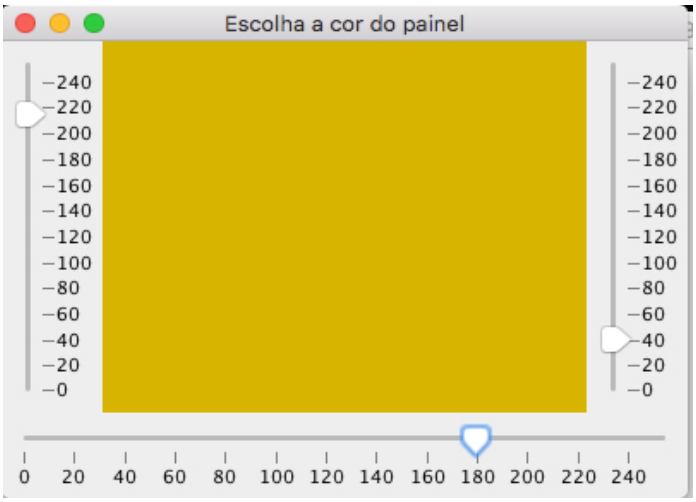


Figura 22.6 JSlder em Ação

Na janela da Fig. 22.6, o usuário selecionou os valores (220, 180, 40) para as intensidades de vermelho, verde e azul, respectivamente.

A classe `SliderCores`, que define o componente exibido pelo programa `UsaSliderCores`, cria as três barras deslizantes e o painel, declara as variáveis que contêm as intensidades das cores e hospedada a classe ouvinte de eventos das barras.

```
1  import java.awt.*;    import java.awt.event.*;
2  import javax.swing.*; import javax.swing.event.*;
3  class SliderCores extends JFrame {
4     private JSlider barraVermelha =
5          new JSlider(SwingConstants.HORIZONTAL,0,255,0);
6     private JSlider barraVerde =
7          new JSlider(SwingConstants.VERTICAL,0,255,0);
8     private JSlider barraAzul =
9          new JSlider(SwingConstants.VERTICAL,0,255,0);
10    int verde, vermelho, azul;
11    private JPanel painel = new JPanel();
12    class Ouvinte implements ChangeListener {...}
13    void detalhaBarra(JSlider barra,Ouvinte ouvinte) {...}
14    public SliderCores() {...}
15 }
```

A construtora de `SliderCores` usa a função `detalhaBarra` para evitar repetição de código na configuração de cada uma das três barras deslizantes, as quais devem ter marcas espaçadas de 20 pixels, devidamente rotuladas e associadas a um ouvinte especificado.

```
1 void detalhaBarra(JSlider barra, Ouvinte ouvinte) {
2    barra.setMajorTickSpacing(20);
3    barra.setPaintTicks(true);
4    barra.setPaintLabels(true);
5    barra.addChangeListener(ouvinte);
6 }
```

A classe dos ouvintes de eventos gerados pelas barras deslizantes é declarada internamente à classe `SliderCores` para facilitar o acesso do método tratador de eventos `stateChanged` às referências das barras deslizantes e às variáveis `vermelho`, `verde` e `azul`, que, respectivamente, detêm as intensidades dessas cores a ser usadas para colorir o fundo do `painel`.

```
1  class Ouvinte implements ChangeListener {
2     public void stateChanged(ChangeEvent e) {
3       JSlider j = (JSlider)e.getSource();
4       int cor = j.getValue();
5       if (j == barraVermelha) vermelho = cor;
6       else if (j == barraVerde) verde = cor;
7            else if (j == barraAzul) azul = cor;
8       Color novaCor = new Color(vermelho,verde,azul);
9       painel.setBackground(novaCor);
10    }
11 }
```

A construtora de `SlideCores` cria um ouvinte a ser compartilhado com as três barras deslizantes, adiciona o painel de exibição na parte central da camada de conteúdo da janela, configura cada uma das barras e as instala no painel, conforme o leiaute definido.

```
1  public SliderCores() {
2     super ("Escolha a cor do painel");
3     Ouvinte ouvinte = new Ouvinte();
4     final Container c = getContentPane();
5     c.add(painel, BorderLayout.CENTER);
6     detalhaBarra(barraVermelha,ouvinte);
7     detalhaBarra(barraVerde,ouvinte);
8     detalhaBarra(barraAzul,ouvinte);
9     c.add(barraVermelha,BorderLayout.WEST);
10    c.add(barraVerde,BorderLayout.SOUTH);
11    c.add(barraAzul,BorderLayout.EAST);
12    setSize(400,300);
13    setVisible(true);
14 }
```

O programa principal ilustrativo barras deslizantes é o seguinte:

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  public class UsaSliderCores {
5     public static void main (String args[]) {
6        JFrame j = new SliderCores();
7        j.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8     }
9  }
```

22.3 Menus

Menus são listas de itens, das quais pode-se selecionar um ou mais itens, os quais podem ser outros menus. Menus são um meio de organizar as funcionalidades de um aplicativo de maneira prática e natural. As principais classes do **swing** utilizadas para definir menus e os seus respectivos métodos que são usados no exemplo a seguir são as seguintes:

- `JMenuBar`: define barra de menu para ser instalada em uma janela. Essa barra pode conter um menu de itens apresentados na primeira linha de um `JFrame`, como a que contém `Informação`, `Cor`, `Fonte` e `Estilo` na Fig. 22.7. Cada um desses itens é um submenu, que quando clicado expande a lista de seus itens. O principal método para gerenciar barras de menu é `void add(JMenu m)`, que serve para adicionar um menu à barra.
- `JMenuItem`: define um item de menu que ao ser clicado gera evento de ação. Os principais métodos para gerenciar itens de menus são:
  - `void setMnemonic(char c)`: associa uma tecla ao item para sua seleção via teclado: teclar *Alt* e a tecla definida por `c`, simultaneamente, tem o mesmo efeito de clicar diretamente no item.
  - `void addActionListener(ActionListener ouvinte)`: adiciona um ouvinte o item.
- `JMenu`: contém os métodos para gerenciar menus:
  - `void add(JMenuItem i)`: adiciona item `i` ao menu.
  - `void add(JCheckBoxMenuItem i)`: adiciona item `i` ao menu.
  - `void add(JRadioButtonMenuItem i)`: adiciona item `i` ao menu.
  - `void addSeparator()`: adiciona um traço separador de itens.
  - `void setMnemonic(char c)`: associa uma tecla ao menu para sua seleção e abertura para exibir seus itens, via teclado.
  - `setToolTipText(String lembrete)`: associa ao menu um lembrete de sua função.

- `JCheckBoxMenuItem`: contém métodos para gerenciar itens de menu que podem ser independentemente selecionados/desselecionados. Os eventos gerados são do tipo `ItemListener`, cujo método `itemStateChanged` é chamado quando um item desse tipo for selecionado ou desselecionado. O ouvinte de eventos é vinculado ao componente via a operação:
  - `addItemListener(ItemListener ouvinte)`
- `JRadioButtonMenuItem`: contém métodos que gerenciam um conjunto de itens de menu de modo que, em um dado momento, apenas um item pode estar ativo. Esses itens devem ser agrupados em conjuntos de seleção exclusiva por meio de objetos da classe `ButtonGroup`. Operações importantes são:
  - `void setSelected(boolean b)`: se `b` for `true`, marca o item como selecionado, mas não dispara eventos.
  - `void addActionListener(ItemHandler i)`: associa ouvinte.

O método de tratamento de eventos usado para cliques em itens de menus depende do tipo de item usado. Em geral, itens que são botões disparam eventos da família `ActionListener`. Para os do tipo `JCheckBoxMenuItem`, usualmente o tratador de evento é do tipo `ItemListener`. Em todos os casos, o evento é disparado pela seleção ou pela desseleção do item.

O programa `UsaMenu`, que ilustra o uso de menus, inicia-se com a janela da Fig. 22.7, a qual exibe uma barra de menu contendo os rótulos dos menus `Informação`, `Cor`, `Fonte` e `Estilo` e uma tela na cor ciano com o texto JAVA.

O usuário pode clicar no botão `x` (vermelho) da janela para encerrar a aplicação, clicar nos itens do menu da barra, como `Informação`, `Cor`, `Fonte` ou `Estilo`, para abri-los. Após um clique em `Informação`, exibe-se a Fig. 22.8.

Figura 22.7 Menu de UsaMenu

Figura 22.8 Menu Informação

O menu `Informação` contém os itens `O Programa`, `Muda Texto` e `Sair`, que quando clicados disparam ouvintes que são objetos do tipo `ActionListener`. Por exemplo, um clique no item `Muda Texto` gera a exibicão da janela da Fig. 22.9, permitindo ao usuário mudar o texto da tela.

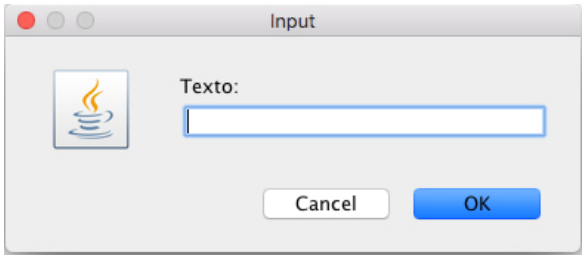


Figura 22.9 Janela de Diálogo

Clique em `Cor` abre um menu com uma lista de itens com nomes de cores, como na Fig. 22.10, da qual pode-se escolher uma para o texto na tela. O item clicado anteriormente aparece marcado,

sendo que, inicialmente, `Preto` aparece selecionado, porque assim foi programado. Os itens são `JRadioButtonMenuItem`, pois apenas uma cor pode ser selecionada por vez. Clique em uma cor dispara o ouvinte do tipo `ActionListener`, o qual está programado para alterar a cor do fonte do texto escrito na tela.

Figura 22.10 Menu Cor

Um clique em `Fonte` abre uma lista de nomes de fontes da qual pode-se escolher um para o texto da tela, conforme mostra a Fig. 22.11. Esses itens são do tipo `JRadioButtonMenuItem` e seus ouvintes são do tipo `ActionListener`.

Figura 22.11 Menu Fonte

Um clique no item `Estilo` da barra de menus abre um menu com uma lista de estilos de apresentação de texto, e.g., `negrito` e `itálico`, da qual pode-se escolher um ou mais para reconfigurar o texto exibido na tela, conforme mostra a Fig. 22.12.

Os itens do menu `Estilo` são do tipo `JCheckBoxMenuItem`,

Figura 22.12 Menu Estilo I

pois mais de um estilo pode ser selecionado de cada vez e seus cliques são tratados por ouvintes do tipo `ItemListener`. Fig. 22.13 mostra seleção simultânea de ambos os itens e o efeito na tela.

Figura 22.13 Menu Estilo II

O programa principal `UsaMenu` segue o modelo padrão que cria uma janela com o botão `x` programado para encerra a aplicação. Toda sua lógica de programação está na classe `Menu`.

```
1 import javax.swing.*; import java.awt.event.*;
2 import java.awt.*;
3 public class UsaMenu {
4    public static void main(String args[]) {
5       JFrame m = new Menu();
6       m.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7    }
8 }
```

A classe central da aplicação, **Menu**, declara inicialmente todos os componentes e tabelas usados no programa. Internamente a essa classe são declaradas as classes dos ouvintes de eventos de clique em itens de menu e itens de estilo, de forma a facilitar-lhes acesso aos atributos e componentes afetados pelos eventos.

```
1  import javax.swing.*; import java.awt.event.*;
2  import java.awt.*;
3  public class Menu extends JFrame {
4    private Color valoresDeCor[] = {Color.black,Color.blue,
5         Color.red,Color.green};
6    private String cores[]={"Preto","Azul","Vermelho","Verde"};
7    private String nomesDeFontes[] = {"Helvetica",
8           "TimesRoman","Monospaced","Courier","SansSerif"};
9    private String nomesDeEstilos[] = "Negrito","Itálico";
10   private JLabel tela;
11   private JMenuBar barra;
12   private JMenu menuDeInfo,menuDeCor,menuDeFonte,menuDeEstilo;
13   private JMenuItem itemPrograma, itemMuda, itemSair;
14   private JRadioButtonMenuItem itensDeCor[],itensDeFonte[];
15   private ButtonGroup grupoDeFontes, grupoDeCores;
16   private JCheckBoxMenuItem itensDeEstilo[];
17   private OuvinteDeMuda ouvinteDeMuda;
18   private OuvinteDePrograma ouvinteDePrograma;
19   private OuvinteDeSair ouvinteDeSair;
20   private OuvinteDeCor ouvinteDeCor;
21   private OuvinteDeFonte ouvinteDeFonte;
22   private OuvinteDeEstilo ouvinteDeEstilo;
23   class OuvinteDeMuda   implements ActionListener {...}
24   class OuvinteDeProg   implements ActionListener {...}
25   class OuvinteDeSair   implements ActionListener {...}
26   class OuvinteDeCor    implements ActionListener {...}
27   class OuvinteDeFonte  implements ActionListener {...}
28   class OuvinteDeEstilo implements ItemListener   {...}
29   public Menu() {...}
30 }
```

Os textos marcados em vermelho acima são refinados a seguir, iniciando pela função construtora, que escreve um texto na tela, cria uma barra de menu e instala os menus a ela associados.

```
1  public Menu() {
2     super("Uso de Menus);
3     Container c = getContentPane();
4     c.setBackground(Color.cyan);
5     tela = new JLabel("Java",SwingConstants.CENTER);
6     tela.setForeground(valoresDeCor[0]);
7     tela.setFont(new Font("TimesRoman",Font.PLAIN,72));
8     c.add(tela,BorderLayout.CENTER);
9
10    barra = new JMenuBar();
11    setJMenuBar(barra);
12
13    "Cria menuDeInfo";
14    barra.add(menuDeInfo);
15
16    "Cria menuDeCor";
17    barra.add(menuDeCor);
18
19    "Cria menudeFonte";
20    barra.add(menuDeFonte);
21
22    "Cria menuDeEstilo";
23    barra.add(menuDeEstilo);
24
25    setSize(500,200);
26    setVisible(true);
27 }
```

A construtora acima inicia-se na linha 2 rotulando a janela e define que sua cor de fundo é ciano. A seguir, da linha 5 a 8, cria-se um rótulo com a palavra `Java` e o insere na camada de conteúdo da janela. Nas linhas 10 e 11, a barra de menu é criada e instalada

na janela. Os passos seguintes são para criar os itens da barra, que são os menus `Informação`, `Cor`, `Fonte` e `Estilo`.

O menu `Informação`, referenciado por `menuDeInfo`, é criado segundo os passos do refinamento de Cria menuDeInfo:

```
1  menuDeInfo = new JMenu("Informação");
2  menuDeInfo.setMnemonic('i');
3  menuDeInfo.setToolTipText("Info sobre o programa");
4
5  itemPrograma = new JMenuItem("O Programa");
6  itemPrograma.setMnemonic('o');
7  ouvinteDePrograma = new OuvinteDePrograma();
8  itemPrograma.addActionListener(ouvinteDePrograma);
9  menuDeInfo.add(itemPrograma);
10 menuDeInfo.addSeparator();
11
12 itemMuda = new JMenuItem("Muda Texto");
13 itemMuda.setMnemonic('m');
14 ouvinteDeMuda = new OuvinteDeMuda();
15 itemMuda.addActionListener(ouvinteDeMuda);
16 menuDeInfo.add(itemMuda);
17 menuDeInfo.addSeparator();
18
19 itemSair = new JMenuItem("Sair");
20 itemSair.setMnemonic('s');
21 ouvinteDeSair = new OuvinteDeSair();
22 itemSair.addActionListener(ouvinteDeSair);
23 menuDeInfo.add(itemSair);
```

O comando da linha 3 acima cria um lembrete da função do menu: se o cursor do *mouse* ficar alguns segundos sobre o item `Informação`, o texto `Info sobre o programa` será exibibo. Os blocos de comandos iniciados nas linhas 5, 12 e 19 instalam os componentes do menu, todos do tipo `JMenuItem`, que respondem a cliques acionando ouvintes do tipo `ActionListener`. Os ouvintes de `itemPrograma`, `itemMuda` e `ItemSair` são os seguintes:

```
class OuvinteDePrograma implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      JOptionPane.showMessageDialog(Menu.this,
         "Uso de Barra de Menus e Menus",
              "Programa", JOptionPane.PLAIN_MESSAGE );
   }
}
class OuvinteDeMuda implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      String s = JOptionPane.showInputDialog("Texto:");
      if (s != null) tela.setText(s);
   }
}
class OuvinteDeSair implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.exit(0);
   }
}
```

No refinamento abaixo de Cria menuDeCor, vale destacar que os itens desse menu são do tipo `JRadioButtonMenuItem`.

```
menuDeCor = new JMenu("Cor");
menuDeCor.setMnemonic('c');
menuDeCor.setToolTipText("Escolha de cor");
itensDeCor = new JRadioButtonMenuItem[cores.length];
grupoDeCores = new ButtonGroup();
ouvinteDeCor = new OuvinteDeCor();
for (int i = 0; i < cores.length; i++) {
   itensDeCor[i] = new JRadioButtonMenuItem(cores[i]);
   menuDeCor.add(itensDeCor[i]);
   grupoDeCores.add(itensDeCor[i]);
   itensDeCor[i].addActionListener(ouvinteDeCor);
}
itensDeCor[0].setSelected(true);
```

E o ouvinte de itens `JRadioButtonMenuItem` do menu `Cor` é o seguinte:

```
class OuvinteDeCor implements ActionListener {
   public void actionPerformed(ActionEvent e ) {
      for (int i = 0; i < itensDeCor.length; i++)
         if (itensDeCor[i].isSelected()) {
               tela.setForeground(valoresDeCor[i]);
               break;
         }
      repaint();
   }
}
```

O menu `Fonte` também é implementado com componentes do tipo `JRadioButtonMenuItem` e agrupados por um objeto da classe `ButtonGroup`, conforme o refinamento Cria menuDeFonte:

```
menuDeFonte = new JMenu("Fonte");
menuDeFonte.setMnemonic('f');
itensDeFonte =
         new JRadioButtonMenuItem[nomesDeFontes.length];
grupoDeFontes = new ButtonGroup();
ouvinteDeFonte = new OuvinteDeFonte();

for (int i = 0; i < itensDeFonte.length; i++) {
   itensDeFonte[i] =
          new JRadioButtonMenuItem(nomesDeFontes[i]);
   menuDeFonte.add(itensDeFonte[i]);
   grupoDeFontes.add(itensDeFonte[i]);
   itensDeFonte[i].addActionListener(ouvinteDeFonte);
}
itensDeFonte[0].setSelected(true);
```

E o ouvinte de itens do menu `Fonte`, também da família dos `ActionListener`, é definido da seguinte forma, na qual vale destacar o cuidado em não alterar o estilo corrente do fonte.

```
class OuvinteDeFonte implements ActionListener {
   public void actionPerformed( ActionEvent e ) {
      int estilo = tela.getFont().getStyle();
      for (int i = 0; i < itensDeFonte.length; i++)
         if (e.getSource() == itensDeFonte[i]) {
            tela.setFont(new Font(
               itensDeFonte[i].getText(),estilo,72));
            break;
            }
      repaint();
   }
}
```

O último menu, definido pelo refinamento Cria menuDeEstilo, implementa itens do tipo `JCheckBoxMenuItem` e permite a seleção simultânea de mais de uma opção.

```
menuDeEstilo = new JMenu("Estilo");
menuDeEstilo.setMnemonic('e');
itensDeEstilo =
         new JCheckBoxMenuItem[nomesDeEstilos.length];
ouvinteDeEstilo = new OuvinteDeEstilo();
for (int i = 0; i < nomesDeEstilos.length; i++) {
   itensDeEstilo[i] =
         new JCheckBoxMenuItem(nomesDeEstilos[i]);
   menuDeEstilo.add(itensDeEstilo[i]);
   itensDeEstilo[i].addItemListener(ouvinteDeEstilo);
}
```

O ouvinte, que nesse caso é do tipo `ItemListener`, deve inspecionar a situação de seleção de todos os itens do menu e combinar os estilos definidos por cada um, e, com o resultado, configurar o estilo do texto na tela.

Os inteiros que representam estilos são valores de 2^n, assim o n-bit de cada inteiro é o que define cada estilo. A soma de

vários estilos forma um inteiro que representa a combinação de todos, que, no presente caso, pode ser **negrito**, **itálico** ou **negrito em itálico**, conforme detalha o seguinte código.

```
1  class OuvinteDeEstilo implements ItemListener {
2     public void itemStateChanged(ItemEvent e) {
3        String fonte = tela.getFont().getName();
4        int estilo = 0;
5        if (itensDeEstilo[0].isSelected())
6               estilo += Font.BOLD;
7        if (itensDeEstilo[1].isSelected())
8               estilo += Font.ITALIC;
9        tela.setFont(new Font(fonte,estilo,72));
10       repaint();
11    }
12 }
```

22.4 Menus sensíveis ao contexto

Um menu sensível ao contexto, também chamado de menu *pop-up*, é implementado por objetos da classe `JPopupMenu`. Esse tipo de menu revela opções específicas do componente gráfico para o qual foi ativado por eventos denominados *eventos de gatilho pop-up*.

O disparador desse tipo de evento depende da plataforma na qual o programa está sendo executado. Na maioria, um clique com botão direito do *mouse* sobre um componente é o gatilho que dispara o *pop-up* do menu, gerando um descritor do evento do tipo `MouseEvent`, que é passado como parâmetro aos métodos tratadores de eventos de ouvintes do tipo `MouseListener`.

Esses tratadores podem aplicar ao parâmetro `e` que lhes for passado o método `isPopupTrigger()` da classe `MouseEvent` para determinar se o evento foi ou não causado por uma ativação de menu `pop-up`. E se esse for o caso, chamar `e.getComponent()`

para obter o componente, e.g. `c`, originador do evento, aplicar o método `popupmenu.show(c,x,y)` da classe `JPopMenu` para fazer o menu `popupmenu` associado ao componente `c` ser exibido na partir das coordenadas `(x,y)` desse componente.

Uma classe que implementa um menu sensível ao contexto deve seguir o seguinte esquema de programa:

```
class PopupMenu extends JFrame{
  private JPopupMenu menu = new JPopupMenu();
 "Construir os itens deø menu e adicioná-los ao menu pop-up";
 "Definir um tratador de evento para cada item do menu";
 "Definir MouseListener para o componente que exibe um
  JPopupMenu quando o evento de gatilho pop-up ocorre";
 "Adicionar o ouvinte MouseListener à janela";
  ...
}
```

onde o refinamento de Adicionar o ouvinte `MouseListener` à janela é tipicamente o seguinte:

```
addMouseListener (
  new MouseAdapter() {
    public void mousePressed (MouseEvent e) {
       checkForTriggerEvent (e);
    }
    public void mouseReleased (MouseEvent e){
       checkForTriggerEvent (e);
    }
    private void checkForTriggerEvent (MouseEvent e){
      if (e.isPopupTrigger() )
         menu.show(e.getComponent(),e.getX(),e.getY());
    }
  }
);
```

22.5 Janelas múltiplas

Tipicamente, uma aplicação consiste em uma janela `JFrame` principal na qual são instalados os componentes gráficos de uma interface. Entretanto, há aplicações que trabalham com uma *Interface de Múltiplos Documentos (multiple document interface - MDI)*, que é formada por uma janela principal contendo outras janelas. A janela principal é frequentemente chamada *janela mãe*. E as janelas internas são chamadas *janelas filhas* ou *janelas internas*.

Esse tipo de interface permite que um mesmo tipo de interação usuário-aplicativo com várias informações que sejam processadas em paralelo. Um exemplo poderia ser um editor de texto que permite a abertura simultânea de mais de um documento, cada um em uma janela separada e que provê para cada uma delas os mesmos recursos para interação.

As classes `JDesktopPane` e `JInternalFrame` do `swing` fornecem suporte para criar interfaces de múltiplos documentos.

```
java.lang.Object
 |
 +--- java.awt.Component
       |
       +--- java.awt.container
             |
             +--- javax.swing.JComponent
                   |
                   +--- javax.swing.JLayeredPane
                         |
                         +--- javax.swing.JDesktopPane
```

Figura 22.14 Hierarquia de `JDestopPane`

A classe `JDesktopPane`, que pertence à hierarquia da Fig. 22.14, permite definir um contêiner para hospedar uma MDI, na qual

pode-se instalar janelas internas do tipo `JInternalFrame`.

Algumas das operações de `JDesktopPane` são:

- `JDesktopPane()`:
  cria uma camada para instalação de objetos `JInternalFrame`.
- `void add(Component c)`:
  adiciona ao contênier `JDesktopPane` o componente `c`, e.g., um objeto `JInternalFrame`.

A classe `JInternalFrame`, que pertence à hierarquia definida na Fig. 22.15, permite exibir componentes gráficos parecidos como uma janela `JFrame`. Em geral, objetos de `JInternalFrame` são adicionados a camadas de conteúdo do tipo `JDesktopPane`. Esses objetos têm muitas das propriedades de janelas, incluindo fechamento, arrastamento, iconização, redefinição de tamanho, título, barras de menu e operações como `add`, `remove` e `setLayout`.

```
java.lang.Object
 |
 +--- java.awt.Component
       |
       +--- java.awt.container
             |
             +--- javax.swing.JComponent
                   |
                   +--- javax.swing.JinternalFrame
```

Figura 22.15 Hierarquia de `JInternalFrame`

Algumas das operações de `JInternalFrame` são:

- `JInternalFrame()`:
  cria uma janela interna.
- `JInternalFrame(String t)`:
  cria uma janela interna com título `t`.

- `JInternalFrame(String t,`
  `boolean alterável, boolean fechável,`
  `boolean maximável, boolean inconificável)`:
  cria uma janela interna com título `t` e as propriedades indicadas pelos demais parâmetros.
- `Container getContentPane()`:
  retorna a camada de contéudo da janela interna.
- `void add(Component c)`:
  adiciona o componente `c` ao contêiner.
- `void remove(Component c)`:
  remove o componente `c` do contêiner.
- `void setLayout(LayouManager m)`:
  define o leiaute da camada de conteúdo da janela interna.
- `void setSelected(boolean foco)`:
  coloca o foco na janela interna, se `foco` for `true`.
- `void setVisible(boolean b)`:
  define a visibilidade da janela interna.
- `void setSize(int largura, int altura)`:
  define o tamanho da janela interna.
- `void setLocation(int x, int y)`:
  define localização da janela interna na camada do componente onde será adicionada.
- `void addInternalFrameListener(`
  `InternalFrameListener ouvinte)`:
  adiciona à janela o ouvinte de seus eventos.

Os ouvintes de objetos `JInternalFrame` são da família da interface `InternalFrameListener`, a qual possui os seguintes tratadores de eventos:

- `InternalFrameOpened(InternalFrameEvent e)`:
  chamado quando a janela interna é aberta.

- `InternalFrameClosing(InternalFrameEvent e)`: chamado em resposta a pedido de fechamento da janela.
- `InternalFrameClosed(InternalFrameEvent e)`: chamado após a janela ter sido fechada.
- `InternalFrameIconified(InternalFrameEvent e)`: chamado devido a clique no botão de iconificar janela.
- `InternalFrameDeIconified(InternalFrameEvent e)`: chamado após clique no botão de deiconificar janela.
- `InternalFrameActivated(InternalFrameEvent e)`: chamado quando a janela é ativada pela presença do *mouse* no seu espaço.
- `InternalFrameDeActivated(InternalFrameEvent e)`: chamado quando a janela foi desativada pela saída do *mouse* de seu espaço.

A correspondente classe adaptadora `InternalFrameAdapter` também está definida para facilitar a programação desses tratadores de eventos.

A criação de uma janela interna segue o seguinte padrão de programação, onde estão destacados os principais passos desse processo:

1. crie uma janela base estendendo a classe `JFrame`;

2. o construtor dessa janela deve:
   - criar um objeto `JDesktopPane` e instalá-lo na camada de conteúdo da janela base;
   - adicionar barra de menu na janela base, se necessário;
   - definir os menus e inseri-los na barra;
   - tornar a janela base visível.

3. definir as classes das janelas internas desejadas, estendendo `JInternalFrame`, cada uma com a definição de seus atributos de tamanho, localização e visibilidade;

4. definir as ações necessárias para criar objetos de cada janela interna, configurando-os adequadamente e inseri-los no objeto `JDestopPane` criado acima;
5. implementar os ouvintes dos itens dos menus, alguns dentre os quais devem criar objetos das janelas internas definidas acima.

O programa desenvolvido a seguir, denominado `UsaJanelaBase`, que ilustra o uso do componente `JInternalFrame`, é uma aplicação que permite a abertura e exibição em janelas distintas do conteúdo de arquivos de texto de forma a facilitar a leitura desses arquivos em paralelo. Inicialmente, o programa `UsaJanelaBase` exibe a tela Fig. 22.16, que apresenta título e barra de menus com o menu *drop-down* `Documentos`:

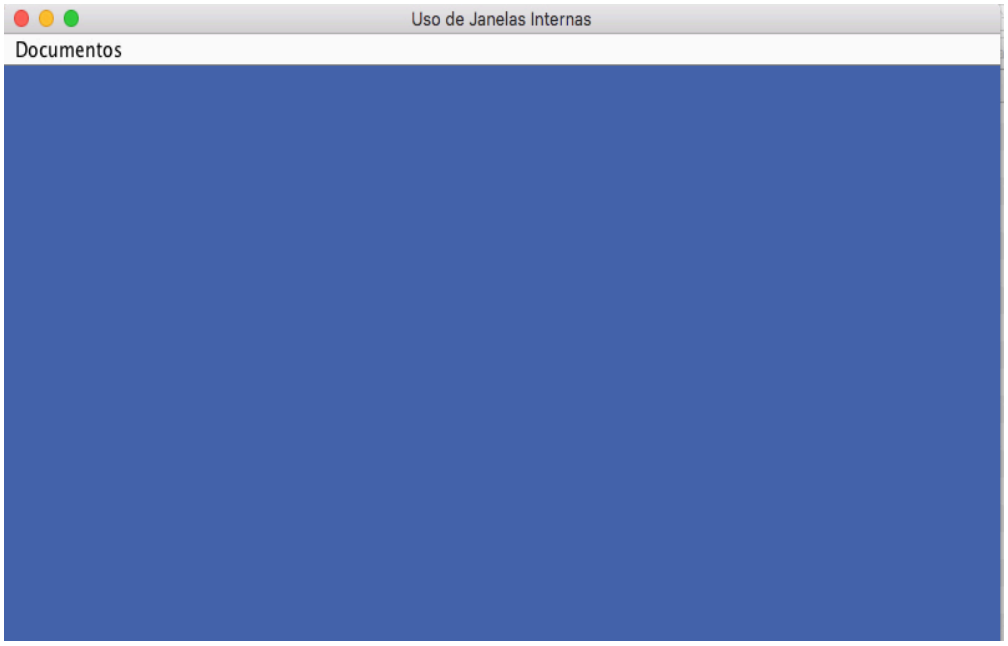


Figura 22.16 DesktopPane para Janelas Internas

Um clique no menu `Documentos` abre o menu exibido na Fig. 22.17, do qual pode-se escolher `Abrir Documento` ou `Encerrar Programa`. Um clique no item de menu `Encerrar Programa` provoca a execução do comando `System.exit(0)`, e clique no item `Abrir Documento`

produz a Fig. 22.18, na qual é solicitado ao usuário o nome do arquivo a ser aberto.

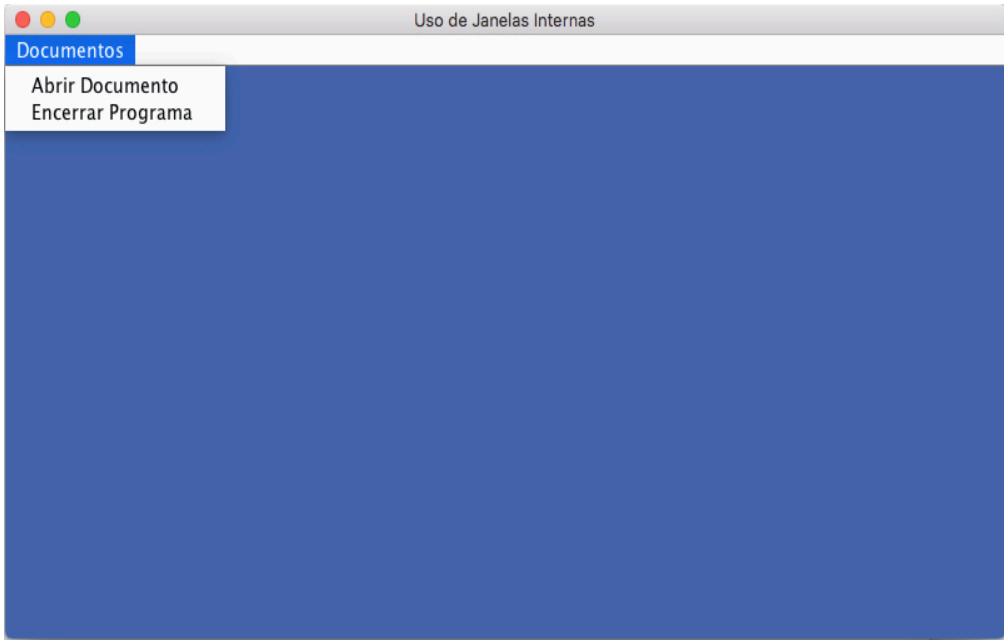


Figura 22.17 Menu Drop-Down

Nesse exemplo, o usuário digitou `apresentação.txt`.

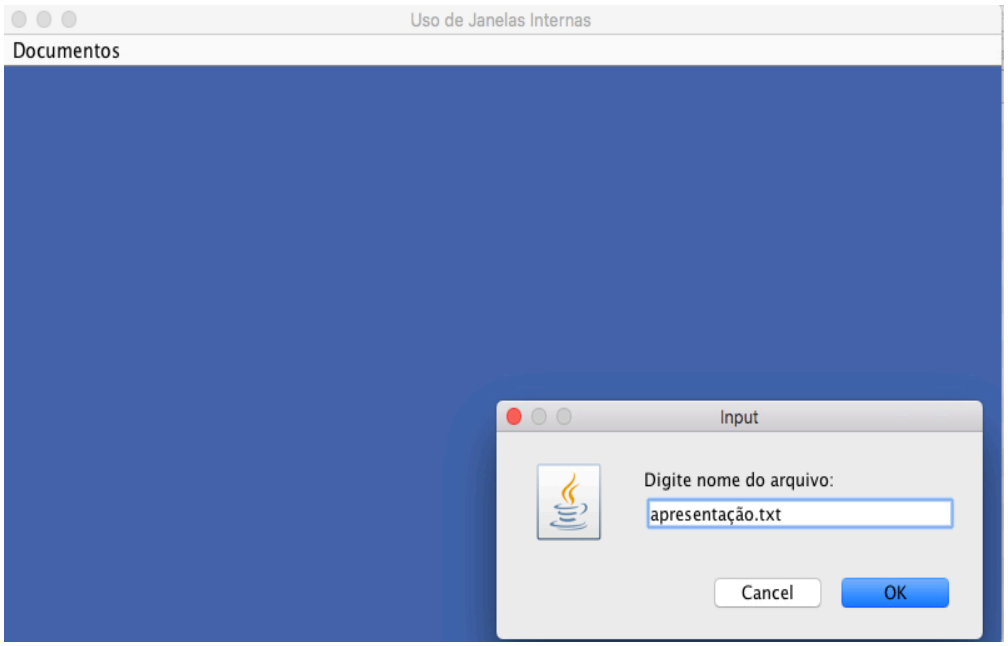


Figura 22.18 Abertura de apresentação.txt

Um clique em `OK` cria a primeira janela interna, abre o arquivo `apresentação.txt` e a exibe seu conteúdo em uma área de texto alocada nessa janela interna, conforme a Fig. 22.19.

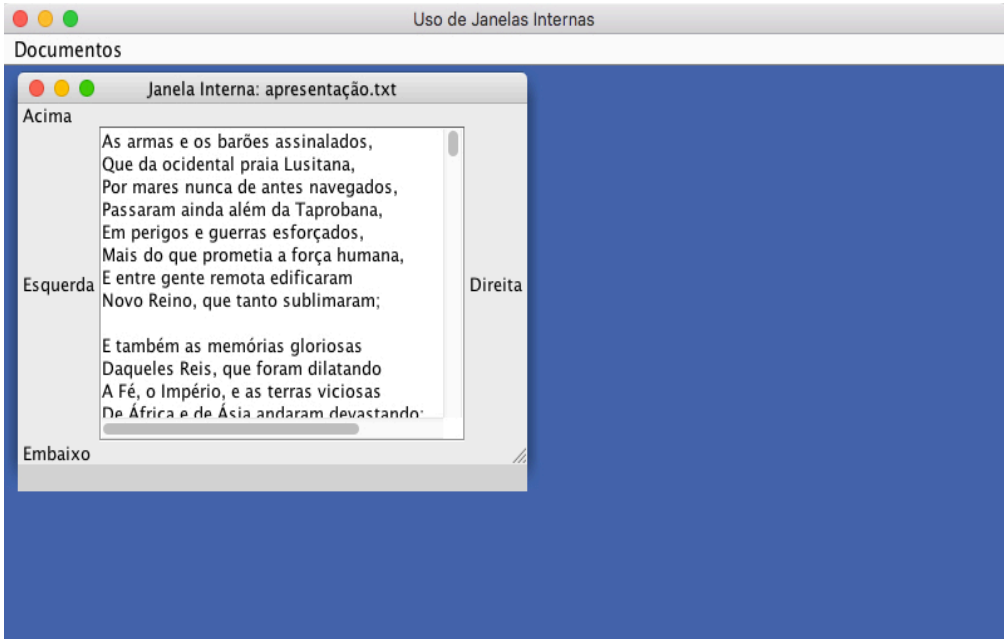


Figura 22.19 Arquivo apresentação.txt na janela gerada

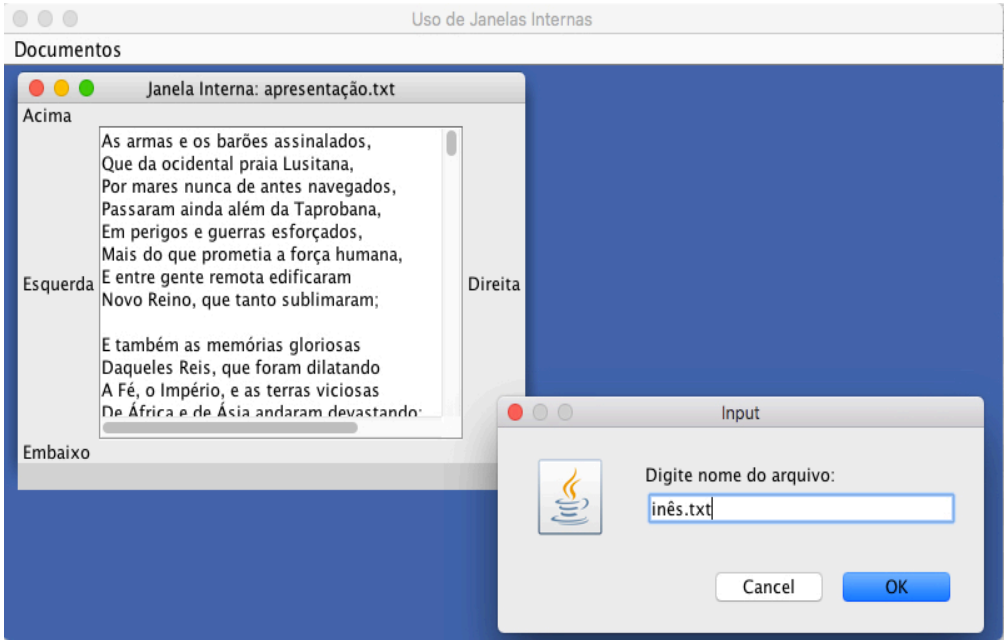


Figura 22.20 Abertura de inês.txt

Um novo clique em **Abrir Documentos** do menu **Documentos** produz a Fig. 22.20, na qual o usuário digitou **inês.txt** seguido de **OK** e produziu a Fig. 22.21, onde uma nova janela interna com o conteúdo do arquivo **inês.txt** é exibido. Essas janelas estão superpostas, mas podem ser livremente, nos limites da janela base, arrastadas para outras posições como o auxílo do ponteiro do *mouse*.

O processo interativo se repete para a criação da terceira janela interna para exibir o conteúdo do arquivo **cilada.txt** e que resulta nas configurações apresentadas nas Fig. 22.22 e Fig. 22.23.

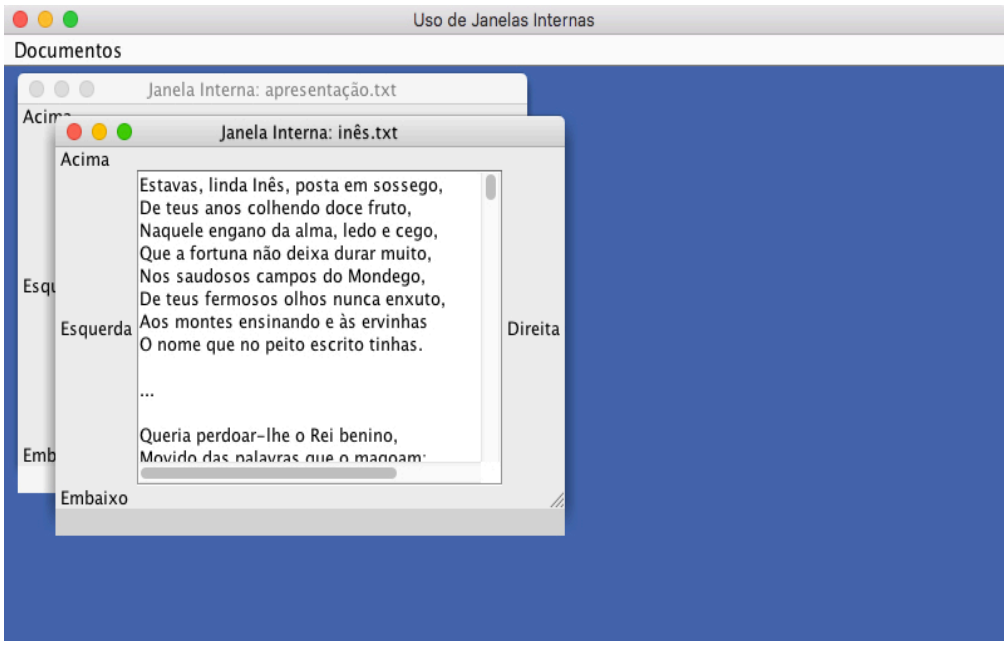


Figura 22.21 Arquivo inês.txt na janela gerada

As janelas internas foram definidas como tendo o leiaute *default* **BorderLayaout**, no qual a região **BorderLayout.CENTER** foi usada para alojar um componente **JTextArea**. As demais regiões desse esquema de leiaute foram apenas marcadas com os rótulos **Acima**, **Esquerda**, **Direita** e **Embaixo**, para mostrar que outros componentes poderiam ser instalados na janela interna da mesma forma que se faz com componentes do tipo **JFrame**.

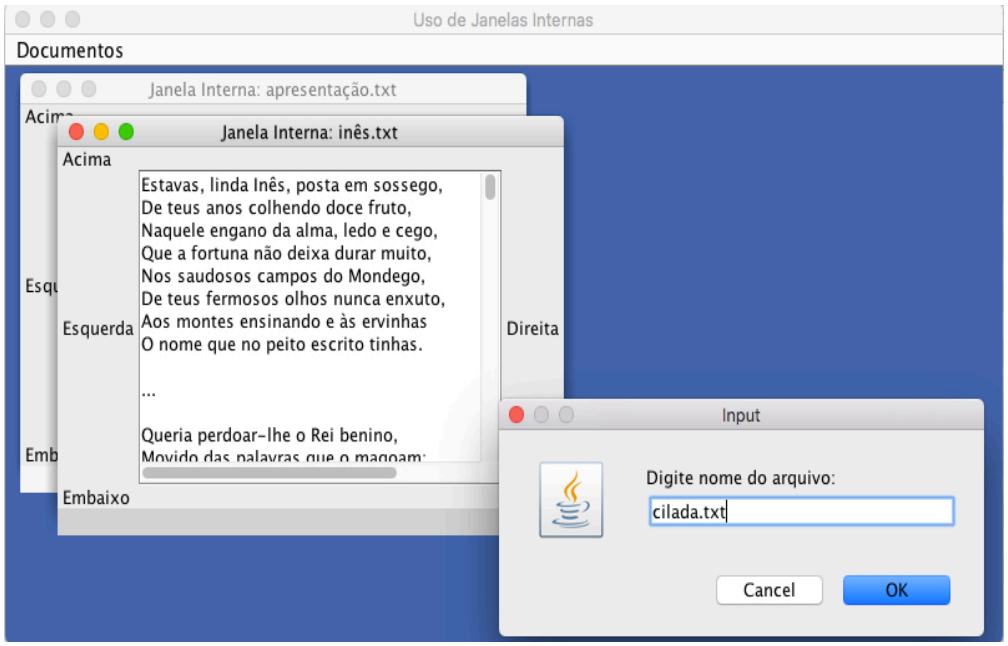


Figura 22.22 Abertura de cilada.txt

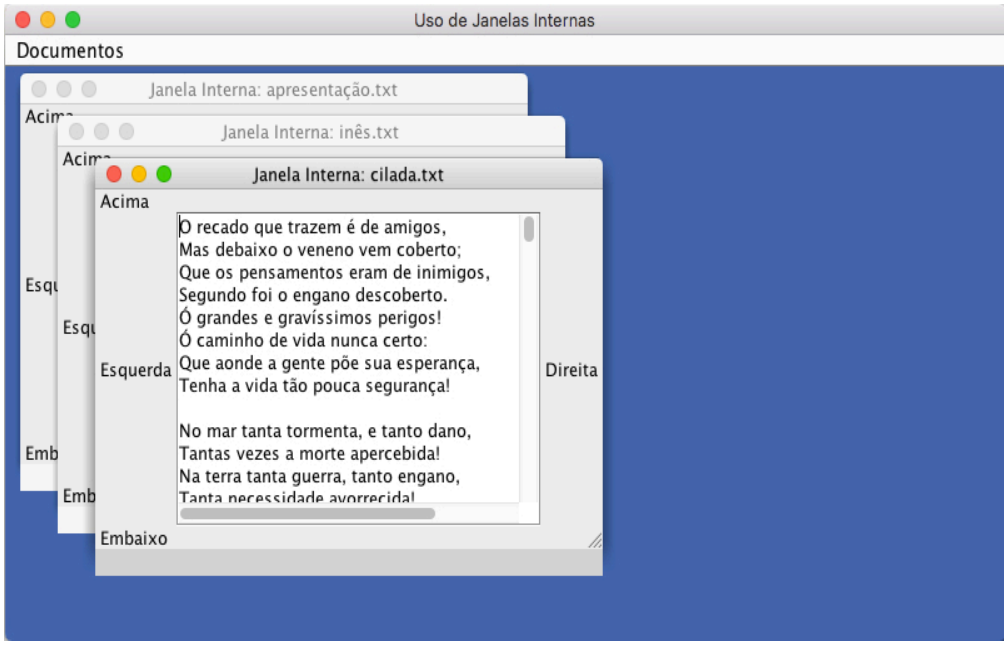


Figura 22.23 Três Janelas Internas Geradas

O programa `UsaJanelaBase` tem a estrutura padrão usada em todos os exemplos deste livro, na qual cria-se a janela principal ou janela base por meio da classe `JanelaBase` da seguinte forma:

```
1 import javax.swing.JFrame;
2 public class UsaJanelaBase {
3     public static void main(String[] args) {
4         JFrame f = new JanelaBase();
5         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6     }
7 }
```

A principal classe da aplicação é a **JanelaBase**, definida a seguir, importa tipos das bibliotecas de Java, conforme o seguinte trecho de código:

```
1 import javax.swing.JInternalFrame;
2 import javax.swing.JDesktopPane;
3 import javax.swing.JMenu;
4 import javax.swing.JMenuItem;
5 import javax.swing.JMenuBar;
6 import javax.swing.JFrame;
7 import javax.swing.JInternalFrame;
8 import javax.swing.JScrollPane;
9 import javax.swing.JTextArea;
10 import javax.swing.JOptionPane;
11 import javax.swing.JLabel;
12 import java.awt.event.ActionListener;
13 import java.awt.event.ActionEvent;
14 import java.awt.BorderLayout;
15 import java.io.BufferedReader;
16 import java.io.FileReader;
17 import java.nio.charset.Charset;
```

Na classe **JanelaBase** destacam-se o uso de **JDesktopPane** como o componente no qual as janelas internas são instaladas e a definição da variável **contador** que é usada para definir a posição das janelas internas dentro da camada de conteúdo do componente base **JDesktopPane**. Internamente a essa classe são definidas as

classes dos ouvintes de cliques em itens do menu instalado na sua barra de menus.

```
1  Importação dos tipos necessários;
2  public class JanelaBase extends JFrame {
3     private JDesktopPane desktop;
4     private static int contador = 0;
5     private JMenuBar menuBar;
6     private JMenu menu;
7     public JanelaBase() { ... }
8     class InternalFrame extends JInternalFrame { ... }
9     protected void criaJanelaInterna(String arquivo)
10                                        throws Exception { ... }
11    class OuvinteDeEncerrar implements ActionListener {...}
12    class OuvinteDeAbrir implements ActionListener { ... }
13
14 }
```

A definição da classe das janelas internas segue o mesmo ritual de criação de classes do tipo `JFrame`:

```
1  class InternalFrame extends JInternalFrame {
2     public InternalFrame(String nome) {
3        super(nome, true, true, true, true);
4        setSize(430,300);
5        contador = (contador + 1) % 3;
6        setLocation(30*contador, 30*contador);
7        setVisible(true);
8     }
9  }
```

A construtora de `JanelaBase`, definida a seguir, cria um componente `JDesktopPane` e o instala em sua camada de conteúdo para receber as janelas internas. Cria-se também a barra de menu com o menu `Documentos`, que contém os itens `Abrir Documento` e `Encerrar Programa`, aos quais são associados os tratadores de

eventos `ouvintesDeAbrir` e `ouvintesDeEncerrar`, respectivamente. Os últimos comandos instalam o menu criado na barra de menus e essa barra na janela interna.

```
1  public JanelaBase() {
2     super("Uso de Janelas Internas");
3     setBounds(50, 50, 800, 600);
4     desktop = new JDesktopPane();
5     setContentPane(desktop);
6
7     menuBar = new JMenuBar();
8
9     menu = new JMenu("Documentos");
10    JMenuItem itemAbrir = new JMenuItem("Abrir Documento");
11    itemAbrir.addActionListener(new OuvinteDeAbrir());
12    menu.add(itemAbrir);
13
14    JMenuItem itemEncerrar=new JMenuItem("Encerrar Programa");
15    itemEncerrar.addActionListener(new OuvinteDeEncerrar());
16    menu.add(itemEncerrar);
17    menuBar.add(menu);
18
19    setJMenuBar(menuBar);
20    setVisible(true);
21 }
```

O `ouvinteDeEncerrar`, que é um `ActionListener`, tem a seguinte implementação, que simplesmente invoca `System.exit(0)` para encerrar a execução da aplicação.

```
1  class OuvinteDeEncerrar implements ActionListener {
2     public void actionPerformed(ActionEvent e) {
3        System.exit(0);
4     }
5  }
```

Ao item **Abrir Documento**, que também é um **ActionListner**, associa-se o ouvinte abaixo, que tem a função de interagir com o usuário para obter o nome do arquivo a ser aberto e criar uma janela interna, na qual são instalados componentes, dentre os quais uma área de texto com o conteúdo do arquivo informado pelo usuário.

O nome do arquivo é obtido via o método **showInputDialog** da classe **JOptionPane**. A reação a clique no botão **cancel** da caixa de diálogo não foi, neste exemplo, implementada e, assim, ela produz o mesmo efeito que **OK**. Caso usuário forneça um nome de arquivo inexistente, o ouvinte insiste re-exibindo a caixa de diálogo. Caso o usuário queira desistir, basta nada digitar e clicar em **OK**.

```
1  class OuvinteDeAbrir implements ActionListener {
2     public void actionPerformed(ActionEvent e) {
3       String arquivo = "?";
4       String msg = "";
5       while (true) {
6         try {
7           arquivo = JOptionPane.showInputDialog(
8                           msg + "Digite nome do arquivo:");
9           if ("".equals(arquivo)) break;
10          criaJanelaInterna(arquivo);
11          break;
12        } catch (Exception ex) {
13             msg = "Arquivo " + arquivo + " não existe.\n";
14        }
15      }
16    }
17 }
```

O método **criaJanelaInterna**, que é chamado pela construtora de **JanelaBase**, recebe como argumento o nome do arquivo de texto a ser exibido na janela interna. Sua primeira ação é tentar abrir um arquivo com o nome fornecido, gerando uma exceção se

não tiver sucesso. Essa exceção não é por ele tratada, sendo seu tratamento delegado ao chamador do método.

O objeto `Charset` usado como parâmetro na chamada da construtora de `FileReader` no código abaixo habilita a leitura correta de caracteres latinos acentuados, necessários nessa aplicação por tratar de textos em Português.

O arquivo é lido linha por linha para montar uma cadeia de caracteres que é usada para preencher uma caixa de texto, a qual é inserida na parte central do painel da janela interna criada.

Os últimos comandos desse método fazem a adição da caixa de texto no componente `desktop` e coloca a janela interna em foco.

```
1  protected void criaJanelaInterna(String arquivo)
2                                              throws Exception {
3     InternalFrame if;
4     JScrollPane texto;
5     Charset charset = Charset.forName("ISO-8859-1");
6     BufferedReader in = new BufferedReader(
7                     new FileReader(arquivo, charset));
8     String s = "";
9     while (in.ready()) s += in.readLine() + "\n";
10    in.close();
11    if = new InternalFrame("Janela Interna: " +
12                                             nomeDoArquivo);
13    texto = new JScrollPane(new JTextArea(s,200,30));
14    if.add(texto,BorderLayout.CENTER);
15    if.add(new JLabel(" Acima"),BorderLayout.PAGE_START);
16    if.add(new JLabel(" Esquerda "),BorderLayout.WEST);
17    if.add(new JLabel(" Direita "),BorderLayout.EAST);
18    if.add(new JLabel(" Embaixo"),BorderLayout.SOUTH);
19    desktop.add(if);
20    if.setSelected(true);  // deve ficar após desktop.add
21 }
```

22.6 Conclusão

Os recursos para programação de menus em interfaces gráficas em Java são extremamente poderosos com dezenas de classes e centenas de métodos. Este capítulo apenas aborda alguns básicos pontos dessa complexa biblioteca, os quais são fundamentais para qualquer iniciante nessa área. Para aprofundar o assunto, recomenda-se a leitura da bibliografia arrolada, em especial os livros dos Deitels.

Exercícios

1. Qual é a vantagem de usar janelas internas no lugar de painéis?

2. Qual é o papel do componente `JDesktoPane` na instalação de janelas internas em uma interface?

Notas bibliográficas

Um dos melhores texto para estudar e aprofundar programação de interfaces gráfica é o livro *Java - Como programar* dos Deitels[10], cujos capítulos 11, 12 e 22 apresentam muitos exemplos ilustrativos de uso dos pacotes `swing` e `awt`. Vale a pena ler.

Capítulo 23

Considerações Finais

Java é uma linguagem orientada por objetos muito popular. Foi projetada na década de 1990 com a proposta de incorporar os conceitos mais modernos de programação modular e privilegiar a segurança, simplicidade, clareza de código e independência de plataforma. Seu projeto foi fortemente influenciado por linguagens como Simula 67, C++, Modula 3, Oberon, Pascal, Lisp, Smalltalk e Eiffel.

Java foi projetada para atender o princípio funcional conhecido como **WORA**: *write once, run anywhere*, permitindo que as aplicações Java possam ser executadas em qualquer plataforma de computação. Para isso, o ambiente de programação de Java é baseado na ideia de se compilar programas Java para uma linguagem intermediária, o *bytecode*, e depois interpretar esse código pela JVM (Java Virtual Machine).

Java possui uma riquíssima biblioteca de interfaces e classes adequada para as mais diversas aplicações e ambientes de programação e que está em constante evolução. Periodicamente uma nova versão do ambiente de programação Java é liberada. O leitor de posse dos conhecimentos dos fundamentos de Java aqui apresentados está preparado para estudar e aplicar essas bibliotecas no desenvolvimento de seus sistemas de computação e acompanhar sua evolução.

Boa sorte e sucesso!

Bibliografia

[1] Arnold, K.; Gosling, J. & Holmes, D. *The Java Programming Language*, Addison-Wesley, Third Edition, 2000, ISBN 0-201-70433-1

[2] Arnold, K.; Gosling, J. & Holmes, D. *A Linguagem de Programação Java, Quarta Edição*, Bookman 2007, ISBN 976-85-60031-64-1.

[3] Arnold, K.; Gosling, J. & Holmes, D. *The Java Programming Language*, 5th Edition, Java Series, Prentice Hall PTR, 992 pages, 2012.

[4] Cardelli, L. & Wegner, P. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17, (4):471-522, December 1985.

[5] Conell, G; Horstmann, C.S. & Cay S. *Core Java*. Makron Books, 1997.

[6] Coad, P.; Mayfield, M. & Kern, J. *Java Design: Building Better Apps and Applets*, Yourdon Press, 2nd Edition, 1999.

[7] Cooper, J. *Java Design Patterns: A Tutorial*. Addison-Wesley, 2000.

[8] Dahl, Ole-Johan & Nygaard, K. SIMULA: an ALGOL-based simulation language. Communications of the ACM, v.9 n.9, p.671-678, Sept. 1966

[9] Dane Cameron. *A Software Engineer Learns Java 8 – The Fundamentals*. Cisdal Publishing. 2014

[10] Deitel, H.M. & Deitel, P.J. *Java: Como Programar*. Sexta Edição. Pearson, Prentice-Hall, 2005.

[11] Descartes, R. *Discurso do Método: Regras para a Direção do Espírito*, Coleção A Obra-Prima de Cada Autor, Editora Martin Claret, 2000

[12] Deremer, F. & Kron, H. *Programming in-the-large versus Programming in-the-small*. In *International Conference on Reliable Software*, pages 114–171, Los Angeles, 1975.

[13] Dijkstra, E. *A Discipline of Programming*. Prentice Hall, 1976.

[14] Doug, Lea. *Concurrent Programming in Java: Design Principles and Patterns*.Second Edition. PAddison-Welsy, 2003.

[15] Ege, R. *Programming in a Object-Oriented Environment*. Academic Press, INC, San Diego, California, 1992.

[16] Ellis, M. & Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[17] Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] Goldberg, A. & Robson, D. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

[19] Goldberg, A. & Robson, D. *Smalltalk the language*. Addison-Wesley, 1989.

[20] Gosling, J.; Joy, B.; Steele, G,; Bracha, G. & Buckkey, A. *The Java^{TM} Language Specification - Java SE 7 Edition*. Oracle, 2011.

[21] Gosling, J.; Joy, B.; Steele, G,; Bracha, G. & Buckkey, A. *The Java^{TM} Language Specification - Java SE 8 Edition*. Oracle, 2014.

[22] Gosling, J.; Joy, B.; Steele, G,; Bracha, G. & Buckkey, A.; Smith, D. & Bierman, G.. *The Java^{TM} Language Specification - Java SE 14 Edition*. Oracle, 2020.

[23] Hehl, M.E. *Linguagem de Programação Estruturada FORTRAN 77*. McGraw-Hill, São Paulo, 1986.

[24] HOARE, C.A.R. *Editorial: The Quality of Software*. In *Software, Practice and Experience*, vol. 2, No 2, pages 103–105, 1972.

[25] Hughes, J.K. *PL/I Structured Programming*. John Wiley & Sons, Inc, 1973.

[26] Ichbiah, J.D. & Morser, S.P. *General Concepts of Simula 67 Programming Language*. 1967.

[27] Johnson, R.E. Frameworks=(Componentes + Patterns). *Communications of the ACM*, october 1997, vol. 40, No. 10.

[28] Kernighan, B. & Ritchie, D. *The C Programming Language (Ansi C)*. Prentice Hall Software Series, Second Edition, 1988.

[29] Laddad, R. *AspectJ in Action: Practical Aspect Oriented Programming*. Manning Publications Co, 2003.

[30] Lintz, B. P. & Swanson, E.B. *Software Maintenance: A User/Management Tug of War* Data Management, pp. 26-30, april 1979.

[31] Lieberherr, K.J. & Doug, O. *Preventive program maintenance in Demeter/Java* (research demonstration). In *International Conference on Software Engineering*, ACM Press, pages 604–605, Boston, MA, 1997.

[32] Liskov, B. & Zilles, S. Programming with Abstract Data Type. *ACM Sigplan Notices*, march, 1974.

[33] Liskov, B. et alii *CLU Reference Manual*. Springer-Verlag, Berlin, 1981.

[34] Liskov, B. et alii *Abstraction and Specification in Program Development*. MIT Press and McGraw-Hill, New York, 1986.

[35] Liskov, B. Data Abstraction and Hierarchy. *ACM Sigplan Notices*, 23,5 (May, 1988).

[36] Martin, R.C. *The Open-Closed Principle*. Disponível na página *www. objectmentor.com/resources/articles/ocp.pdf*. Acesso: 28/02/2005.

[37] Martin, R.C. *Design Principles and Design Patterns*. Disponível na página *www.objectmentor.com*. Acesso: 28/02/2005.

[38] Martin, R.C. *The Liskov Substitution Principles*. Disponível na página *www.objectmentor.com*. Acesso: 28/02/2005.

[39] Mirador, *Enciclopédia Mirador Internacional*. Encyclopedia Britannica do Brasil Publicações Ltda, Vol 15, 1986.

[40] Myers, G. J. *Object-Oriented Software Construction*. Second Edition. Prentice Hall, 1997.

[41] Myers, G. J. *Reliable Software Through Composite Design*. Petrocelli/Charter, 1975.

[42] Myers, G. J. *Composite/Structured Design*. Van Nostrand Reinhold Company, 1978.

[43] Oracle, *Java Language Changes and JDK 14 Documentation*. `https://docs.oracle.com/en/java/javase/14/`, 2020.

[44] Ossher, H. & Tarr, P. *Hyper/J: Multi-Dimensional Separation of Concerns for Java*. In *Proceedings of the 22nd international conference on Software Engineering*, pages 734–737. ACM Press, 2000.

[45] Ossher, H. & Tarr, P. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, 2001.

[46] Parnas, D. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[47] Rumbaugh, J.; Jacobson, I. & Booch, G. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, 1999.

[48] Rumbaugh, J.; Jacobson, I. & Booch, G. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.

[49] Scott, W. A. *Análise e Projeto Orientados a Objeto*. Volume 2, IBPI Press, Livraria e Editora Infobook S.A., 1998

[50] Tirelo, F.; Bigonha, R.S.; Valent, M.T.O & Bigonha, M.A.S. Programação Orientada por Aspectos, IN *JAI 2004*, SBC, Salvador.

[51] Strachey, C. *Fundamental concepts in programming languages*. Lecture notes for International Summer School in Computer Programming. Copenhagen, August 1967.

[52] Strachey, C. *Fundamental concepts in programming languages*. Higher-Order and Symbolic Computation. Copenhagen, April 2000.

[53] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1997.

[54] von Staa, A. *Programação Modular*. Editora Campus, 2000.

[55] Wexelblat, R.L. (ed.) *History of Programming Languages*. Academic Press, 1981, pp 25-74.

Índice Remissivo