PROGRAMAÇÃO MODULAR EM JAVA

Volume I

A Linguagem Java

Primeira Edição

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

Bigonha et al.
PROGRAMAÇÃO MODULAR EM JAVA: Volume I- A Linguagem Java/ Roberto da Silva Bigonha, –
Belo Horizonte, MG: Editora Bigonha, 2010.

Bibliografia.
ISBN 88-888-8888-8

1. Ciência da computação 2. Engenharia da computação
4. Informática 5. Sistemas de informação I. Título.

99-9999 CDD-999

Índice para catálogo sistemático:

1. Computação : Linguagens de Programação
Engenharia de Software 004???

PROGRAMAÇÃO MODULAR EM JAVA

Volume I

A Linguagem Java

Primeira Edição

Roberto da Silva Bigonha

Belo Horizonte, MG, Brasil

Gerente Editorial:
Roberto Bigonha

Editor de Desenvolvimento:
Roberto Bigonha

Produtor Editorial:
Roberto Bigonha

Supervisor de Produção Editorial:
Roberto Bigonha

Copidesque:
Roberto Bigonha

Revisão:
Roberto Bigonha

Composição:
Roberto da Silva Bigonha

Capa:
Roberto Bigonha

Impresso no Brasil.
Printed in Brazil.

Rua Oliveira Pena, 126
B. São José — Pampulha
31.275-130 Belo Horizonte – MG

Dados internacionais de Catalogação na Publicação Câmara Brasileira do Livro Belo Horizonte, MG
Programação Modular em Java
Volume I- A Linguagem Java
Roberto S. Bigonha
Belo Horizonte
ISBN ??-???-????-?

Índice para catálogo sistemático:
1. Computação : Fundamentos :
Linguagens de Programação

Sumário

Prefácio

Este livro foi escrito para ser utilizado por 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. O livro está organizado em cinco volumes: **Volume I**: A Linguagem Java; **Volume II**: Princípios de Projeto; **Volume III**: Métricas de Modularização; **Volume IV**:Programação Orientado por Aspectos; **Volume V**: Estruturas de Dados Fundamentais.

Volume I trata da apresentação da linguagem Java, suas estruturas básicas de tipos e de controle de execução, destacando-se seus recursos para programação modular. Parte-se do princípio de que sem o recurso lingüístico adequado 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 específica linguagem moderna como meio de comunicação e de implementação dos exemplos é, portanto, imperativo para dar clareza aos conceitos e idéias defendidos. A escolha de Java certamente é arbitrária, outras linguagens com recursos para programação modular existem e poderiam ter sido usadas.

No Volume II, Princípios de Projeto, são apresentados o contexto no qual os conceitos e técnicas aqui discutidos estão inseridos, a motivação e recursos necessários para o desenvolvimento de software modular, o impacto da modularidade no custo de manutenção de programas e em outros fatores de qualidade de software. Discute-se também o papel das metodologias de desenvolvimento de software, suas vantagens e desvantagens. Os conceitos e técnicas da programação modular são apresentados, discutidos e avaliados. Particular atenção é dada a estilo e padrões de programação. Práticas condenadas são expostas, e princípios e critérios de projeto de programas modulares de boa qualidade são enunciados e fundamentados. Este volume conclui-se com a apresentação de um conjunto de mandamentos da programação de boa qualidade, definidos segundo os princípios que balizaram este trabalho.

Volume III, Métricas de Modularização, discute técnicas para medir o grau de modularização de um sistema e critérios de qualidade. Volume IV, Programação Orientado por Aspectos, apresenta uma introdução à Programação Orientada por Aspectos e como esta nova tecnologia pode ser usada para produção de programas modulares. Volume V, Estruturas de Dados Fundamentais, mostra detalhes da implementação modular das estruturas de dados fundamentais, usando as técnicas descritas nos volumes anteriores desta série.

Roberto da Silva Bigonha

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, a Fernando Magno Pereira Quintão os exemplos de interface gráfica e a Luciano Capanema Silva, Thales Evandro Simas Júnior e Abdenago Alvim, alunos do Curso de Especialização em Informática, a indicação de erros remanecentes.

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.

Os 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. Nestes 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 isto, 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: `class, instanceof, throws, native, interface, abstract, synchronized`
- Em módulos: `extends, implements, package, import`
- Reservadas para futuro uso: `cast, const, future, generic, goto, inner, operator, outer, rest, var`

1.2 Programa em Java

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 de Java

Os tipos de dados de Java são classificados em primitivos ou referências. Os tipos primitivos são os tipos numéricos inteiros (`byte`, `char`, `short`, `int`, `long`), os numéricos fracionários (`float`, `double`) e os booleanos (`boolean`). Os tipos referência compreendem classes, interfaces e arranjos.

Variáveis de tipo primitivo obedecem ao modelo **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` definido pelas operações sem operandos ou constantes `true` e `false`, a operação unária negação (`!`) e as operações binárias conjunção condicional (`&&`), dis-

junção 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 estas 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 como Java, que permite efeito colateral de expressões, deve-se dar atenção a possibilidade de expressões serem ou não avaliadas. O trecho de programa abaixo, que ilustra este fato, atinge seu fim com `b = false` e `i = 11`:

```
boolean b, b1 = true, b2 = false, b3 = true;
int i = 10;
b = !(b1 | b2 && (i++= 10)) & (i++ == 10);
```

1.5 Tipos Numéricos

Os tipos numéricos inteiros são `byte`, `char`, `short`, `int` e `long`. Todos estes 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 (`=<`), 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 para a esquerda (`<<`). O deslocamento aritmético 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(`>>>`)

Tipo `int`

O tipo `int` compreende valores inteiros do intervalo $[-2.147.483.648, 2.147.483.647]$, representados em 32 bits. As constantes do tipo têm formato **decimal**, escrito como sequências de algarismos decimais, **octal**, que são sequências de algarismos decimais iniciadas por `0` ou **hexadecimal**. que são iniciadas por `0x` e formadas por dígitos hexadecimais.

Tipo `long`

O tipo `long` é o tipo dos inteiros representados em 64 bits 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

$$[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807].$$

Tipo `byte`

O tipo `byte` compreende os valores inteiros no intervalo $[-128, 127]$, representados em 8 bits. O tipo não tem constantes e nem operações, mas as constantes inteiras do tipo `int` ou `char` podem ser usadas, sem necessidade de conversão, desde que os valores caibam em 8 bits.

O trecho de código abaixo mostra a necessidade de conversão dos resultados para ter os resultados em 8 bits:

```
int  x = 1;
byte b1 = 1, b2 = 2;        // Correto
byte b3 = b1 + b2 ;         // Erro de compilação
byte b4 = (byte) (b1 + b2); // Correto
byte b5 = (byte) (512 + 3); // Truncamento: b5 recebe 3
byte b6 = x;                // Erro de compilação
byte b7 = (byte) x;         // Correto
```

Tipo `short`

O conjunto de valores do tipo `short` são as constantes inteira no intervalo $[-32.768, 32.767]$, representadas em 16 bits. Análogo ao tipo `byte`, este tipo também não possui constantes próprias, mas pode-se usar as constantes `int` ou `char`, sem necessidade de conversão explícita, desde que os valores caibam em 16 bits, formato complemento de 2.

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

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

Tipo `char`

O tipo `char` é um tipo numérico caracterizado pelos inteiros no intervalo $[0, 65.535]$, representados com 16 bits. 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: $'$A $'$, $'$a $'$

- Caracteres de escape:

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

- Seqüência de escape octal: somente os oito bits menos significativos, isto é, `'\000'` a `'\377'`
- Seqüência 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); // Correto
char u = (char) (x - y); // Erro de lógica?
```

O tipo `char` tem um comportamento dual, pois conforme o contexto é tratado como numérico ou charactere, conforme mostra o programa:

```
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
```

Tipo `float`

Os 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, e.g., `1E1F`, `1e1f`, `2.F`, `2f`, `.3f`, `3.14f`, `6.02E+23f`.

Tipo `double`

Os valores de tipo `double` estão no intervalo $[-1.7E308, 1.7E308]$ aproximadamente, com 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, 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  x = 6.5f;
float  y = 6.5;            // Errado: 6.5 é double
float  z = (float)6.5;
byte   b = 1;
float  t = b;
```

Erros e Conversões Automáticas

Operações com inteiros **nunca** indicam ocorrências de estouro de valor, para cima (*overflow*) e 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`, são automaticamente promovidos a `int`, se pelo menos um dos operandos da operação for um `int`, e a operação é realizada com 32 bits, produzindo resultado do tipo `int`.

Numa operação de inteiros, se pelo menos um dos operandos for `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 = 1000000000000000000;
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 em Java por meio da declaração de enumerações ou então por meio do modificador `final`, o qual informa ao compilador que a "variável"declarada não pode ter seu valor alterado após sua inicialização. No exemplo a seguir, a classe `Naipe1` e a enumeração `Naipe2` ilustram as duas formas de declaração de constantes:

```
1  class Circle {
2    public static final double PI = 3.1416;
3  }
4  class Naipe1  {
5    final static int PAUS    = 1;  final static int ESPADAS = 2;
6    final static int OUROS   = 3;  final static int COPAS   = 4;
7  }
8  enum Naipe2{
9     PAUS, ESPADAS, OUROS, COPAS;
10 }
```

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. 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.

```
1  public class UsoDeConstantes {
2    public static void main(String[] args) {
3       int x = Naipe1.OUROS;
4       Naipe2 y = Naipe2.OUROS;
5       double raio, area;
6       .......
7       area  = Circle.PI * raio * raio;
8       int y = Naipe1.OUROS + Naipe2.OUROS;   // Erro de Compilação
9    }
10 }
```

1.7 Expressões

As expressões em Java podem ser aritméticas, lógicas ou condicionais, podem ter efeito colateral e são formadas por operandos, operadores unários, binários e ternários. Pares de parêntesis podem ser usados para dar clareza ou definir ordem de aplicação de operadores.

Nas tabelas que se seguem os termos `e`, `a` e `b` denotam expressões (operandos), `v`, uma variável na qual pode-se armazenar um valor, e `t` representa um tipo de dado.

Os operadores de um único operando são:

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`	dá o valor do numérico em `v` após incrementar o seu conteúdo
`--v`	dá o valor do numérico em `v` após decrementar o seu conteúdo
`v--`	dá o valor de numérico em `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

Os operadores binários aplicáveis a operandos do tipo `boolean` são:

operação	Valor
`a & b`	*and* dos valores 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`

Os operadores binários aritméticos usados em expressões java são:

operação	Valor
`a * b`	multiplicação
`a / b`	divisão de `a` por `b`
`a % b`	resto da divisão do inteiro `a` pelo inteiro `b`
`a + b`	adição ou concatenação de `a` e `b`
`a - b`	subtração de `a` e `b`

O operador de atribuição e os de comparação aplicáveis a valores de todos os tipos, sejam numéricos, booleanos ou referências, são:

operação	Valor
`a = b`	valor de `b` e colateralmente atribui este à variável `a`
`a == b`	`true` se valor de `a` é igual ao valor de `b`
`a != b`	`true` se valor de `a` é diferente do valor de `b`

Os seguintes operadores de comparação são aplicáveis somente a valores numéricos inteiros ou fracionários:

operação	Valor
`a < b`	`true` se `a` for menor que `b`
`a > b`	`true` se `a` maior que `b`
`a <= b`	`true` se `a` menor ou igual `b`
`a >= b`	`true` se `a` maior ou igual `b`

Na tabela abaixo, são definidos operadores de deslocamento e operadores de lógica bit-a-bit, aplicáveis a expressões do tipo inteiro. Os termos `a` e `b` são expressões do tipo `int` ou `long` e `c` deve ser do tipo `int` ou de inteiro menor:

operação	Valor
`a << c`	`a` com `c` bits deslocados para a esquerda
`a >> c`	`a` com `c` bits deslocados aritmeticamente para a direita
`a >>> c`	`a` com `c` bits deslocados logicamente para 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`

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.

Java possui um operador de três operandos (`? :`), usado para definir expressões condicionais, da forma:

```
e ? b : c
```

onde `e` é uma expressão booleana e `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:

```
x = (i > 0 ? 10 : 20);
```

atribui o valor 10 a `x` se `i<0`, ou valor `20`, caso contrário.

As prioridades dos operadores e as respectivas regras de associatividade estão definidas em ordem decrescente na tabela abaixo:

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

Independentemente das prioridades definidas na tabela acima, e da ordem de execução das operações, os operandos de uma expressão são avaliados da esquerda para a direita. O programa

```
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 }
```

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 = 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
```

que 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 vale `2`, é obtido. Em seguida, pela ordem, computam-se `i=3`, `v[3]`, `i+=4` e `[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]`.

Lembre-se que os operandos de `&&`, `||`, `?:` e `,` são avaliados à medida que sejam necessários, e abortam a avaliação da expressão tão logo seja possível inferir o seu resultado. Como expressões podem ter efeito colateral, isto é, podem modificar o estado do programa além de produzir um valor, e nem toda expressão é avaliada, muito cuidado é recomendado no uso deste tipo de expressão.

Por exemplo, no programa

```
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 }
```

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. Se `v` é uma referência ao arranjo, então `v[i]` é uma referência ao seu elemento de ordem `i+1`, desde que valor de `i` situe-se no intervalor $[0, \texttt{v.length-1}]$. O tamanho do arranjo alocado é definido no momento da criação do arranjo. Índices de arranjos são do tipo `int`. Valores de inteiros menores são automáticamente promovidos a `int`. Não se pode indexar arranjos por inteiros `long`. Indexação fora de limites gera a exceção `IndexOutOfBoundsException`.

Em uma referência a arranjo, a expressão anterior aos `[ ]` 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 serão liberados automaticamente, quando não forem mais necessários.

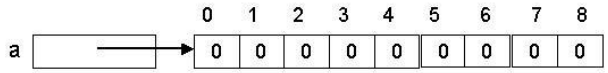
Na criação de um objeto arranjo, especificam-se **tipo de seus elementos**, **número de níveis de aninhamento de arranjos** e o **tamanho** de pelo menos da primeira dimensão. As dimensões omitidas são sempre as mais à direita, isto é, se omite-se a i-th dimensão, todas as dimensões maior que i também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

Todos os elementos de um objeto arranjo alocado 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`.

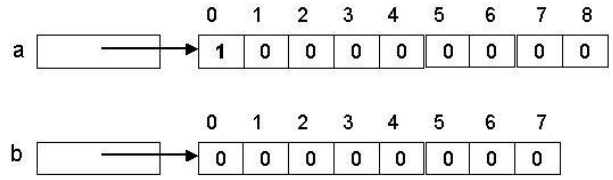
A declaração `T[ ] v = new T[tamanho]`, onde `T` é um nome de um tipo válido em Java, tem o efeito de:

- definir `v` como uma referência para arranjos de elementos do tipo `T`;
- alocar um objeto arranjo de comprimento `tamanho` de elementos do tipo `T`;
- iniciar o objeto arranjo conforme o tipo `T`;
- atribuir o endereço da área alocada à referência `v`.

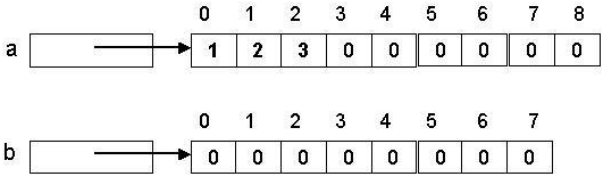
Por exemplo, a declaração `int a [ ] = new int[9]` produz:



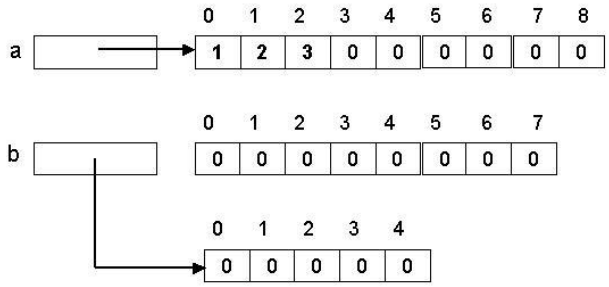
As declarações `int [ ] a = new int[9]; int [ ] b = new int[8]` produzem



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 seguinte configuração:



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 para



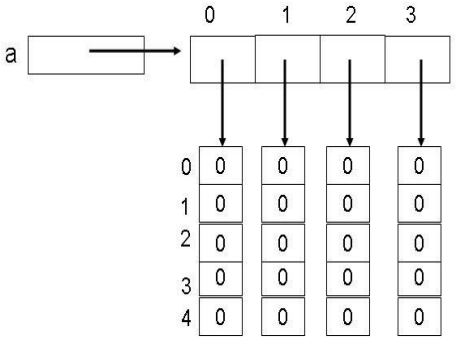
e

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 esta área. Áreas alocadas para qualquer objeto são liberadas automaticamente quando o objeto não for mais referenciado no programa.

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];
```

Esta 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 deste vetor. A seguir aloca 4 vetores, todos zerados, de 5 inteiros cada um e inicia cada posição do vetor apontado por `a` com o endereço de cada um destes vetores de inteiros. O resultado é o equivalente a uma matriz $6X5$, conforme ilustrado pela figura:

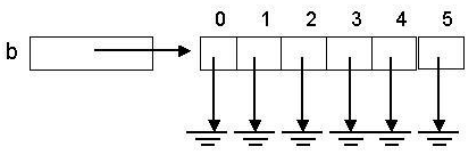


A alocação de arranjo de arranjos pode ser parcial, isto é, pode-se omitir, no operando de `new`, a dimensão de arranjos da direita para a esquerda, exceto a do primeiro índice.

Considere o seguinte trecho de programa:

```
int [ ][ ] b;
b = new int[6][ ];
```

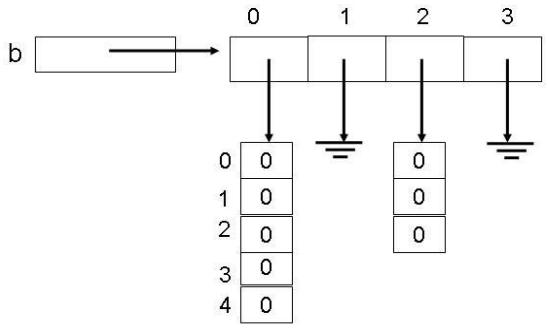
onde somente um arranjo de referências zeradas para vetores de inteiros é alocado:



Se, a seguir, executam-se os comandos:

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

tem-se uma *matriz* cujas colunas são desiguais em tamanho:



Em uma declaração contendo uma única referência, como em

```
char c [ ] [ ];
char [ ] c [ ];
char [ ] [ ] c;
```

a posição dos colchetes em relação ao elemento declarado não faz diferença. Por exemplo, as três declarações acima 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:

```
short [ ] varal, i[ ], j, k[ ][ ];
```

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, `varal`, `i[ ]`, `j` e `k[ ][ ]` são referências para arranjo de `short`. Por conseguinte, `i` é uma referência para um arranjo de arranjo de `short`, e `k` é uma referência para arranjo de arranjo de arranjo de `short`.

Arranjos podem ser iniciados no momento de sua declaração com valores diferentes dos valores padrões definidos pelo operador `new`. O valor de iniciação do arranjo pode ser definido por uma expressão de iniciação de arranjo, que consiste em uma lista. delimitada por 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.

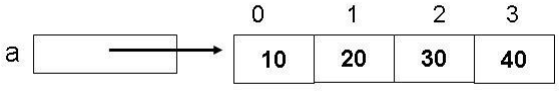
O operador de alocação de objeto `new` pode ser omitido. Assim a declaração

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

é equivalente a

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

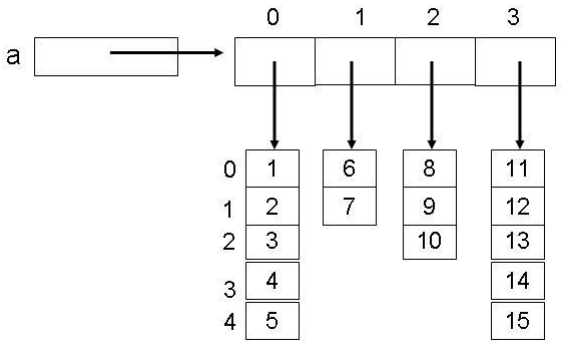
e produz o efeito indicado pela figura:



No caso de arranjos multidimensional, deve-se codificar iniciador dentro de iniciador. Um exemplo de alocação e iniciação de um arranjo bi-dimensional 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 figura:



A operação `a.length` informa o número de elementos alocados para o objeto arranjo diretamente apontado pela referência `a`. 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`, como ilustra o programa:

```
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 comandos de Java são atribuição, bloco, condicionais (`if` e `switch`), de repetição (`while`, `do-while`, `for`, `for` aprimorado), retorno de método (`return`), controle de repetição (`continue` e `break`), de controle de exceções (`try-catch-finally`) e de controle de concorrência (`synchronized`).

Atribuição

O comando de atribuição é 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 obter um endereço de memória, obter o valor associado a este 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 a posição do arranjo atualizada.

Bloco

O bloco é uma estrutura que permite reunir um grupo de comandos e um escopo local de declarações. Deve ser usado sempre que se quiser codificar vários comandos onde apenas um for permitido ou se desejar delimitar o escopo de certas declarações.

Um comando bloco tem um dos formatos:

- `{ declarações e comandos }`
- `{ comandos }`

Por razões de legibilidade, Java não permite a ocultação de nomes em escopo aninhados, como ocorre com `j` na função `main` abaixo:

```
public static void main(String[] args) {
   int i = 10, j ;
   if (i == 10) {
      int j = 100;  // Erro de compilação
      i += j;
   }
   j = i*10;
}
```

Todavia, como o escopo de uma declaração em Java inicia-se somente no ponto em que ocorre, estendendo-se até o fim do bloco, a função `main` abaixo é válida:

```
public static void main(String[] args) {
   int i = 10;
   if (i == 10) { int j = 100; i += 10; }
   int j;
   j = i*10;
}
```

Comando `if`

O comando `if`, que permite escolher entre duas possibilidades de continuação 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-exemplo 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        char c = s.charAt(3);
5        System.out.print("Quarto caractere de Arg[0]");
6        if (s.length > 3 && a.charAt[3] == 'A')
7             System.out.print(" ");
8        else System.out.print(" é indefinido ou não ");
9        System.out.print("é um A");
10    }
11 }
```

Se a ativação do programa `UsoDoIf` for

```
C:>java UsoDoIf M1gA---2kdjgadsdwHH88AAA
```

o texto impresso será:

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

Comando `switch`

O comando `switch`, que permite fazer uma seleção de 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 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 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`.

O programa abaixo imprime `"x = 3"`. Confira, no programa abaixo, que o fluxo de execução passa pelos comandos associados aos casos 2, 3 e 4 antes de deixar o `switch` pela execução de um 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 }
```

Comando `while`

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

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

Na primeira forma, *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. A 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 efeito de imprimir os valores do vetor `a`, iniciando pelo primeiro valor diferente de zero. 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 teste da condição de parada é feito após cada execução do corpo do comando. Neste caso, é garantida a execução do corpo do comando pelo menos uma vez.

O trecho de programa a seguir produz o mesmo resultado do programa `While` acima, embora pareça um pouco mais complicado:

```
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[h]);
9             h++;
10       } while ( h < a.length );
11    }
12 }
```

Comando `for`

O comando de repetição `for` tem a forma:

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

A execução do comando `for` inicia-se com a avaliação da expressão *iniciação*. A seguir, a expressão *condição* é avaliada, e se resultar `true`, o comando *corpo* é executado. Após o término da execução do *corpo*, a expressão *avanço* é avaliada e em seguida reavalia-se a expressão *condição* do comando `for`, e se ainda continuar com 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`, ponto em que a execucão prossegue com o próximo comando. A execução comando `for` também pode ser encerrada pela execução de um comando `break`.

A expressão de iniciação do comando `for` pode conter declaração de variáveis, como é o caso de `int j` no programa `VarDeControle`, que inicia o arranjo `a` com inteiros de 1 a 9, acha a soma destes inteiros e imprime a valor da soma e o conteúdo de `a` do fim para o seu início:

```
1  public class VarDeControle {
2     public static void main(String[ ] args) {
```

```
3        int i, s = 0;
4        int a [ ] = {1,2,3,4,5,6,7,8,9};
5        for (i = 0; i < a.length ; i++) s += a[i];
6             System.out.println("Soma de a = " + s);
7        for (int j = a.length - 1, j >= 0; j--) {
8             System.out.println(""a[" + j + "]= " + a[j]);
9     }
10 }
```

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`. Por exemplo, o comando, com duas variáveis de controle `i` e `j`,

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

é válido, se `i` e `j` não tiverem sido declarados em um bloco envolvente.

Por outro lado, o comando `for`

```
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]));
}
```

é inválido, porque há mais de uma declaração na expressão *iniciação*.

Observe que o programa

```
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+"]= "+(a[i]+a[j]));
7        System.out.println("a["+i +"]+a["+j+"]= "+(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+"]= "+(a[i]+a[j]));
10    }
11 }
```

apresenta os erros de se ter duas declarações na expressão *iniciação* dos dois comandos `for` e de indevidamente ocultar as variáveis `i` e `j`.

O conserto a ser feito para fazer o programa ForErrado funcionar é:

```
1  public class For2 {
2    public static void main( String [] args) {
3       int a [ ] = {1,2,3,4,5,6,7,8,9};
4       for (int i=0, j=a.length-1 ; i<a.length ; i++,j--)
5            System.out.println("a["+i +"]+a["+j+"]= "+(a[i]+a[j]));
6       for (int i=0, j=a.length-1 ; i<a.length ; i++,j--)
7            System.out.println("a["+i +"]+a["+j+"]= "+(a[i]+a[j]));
8    }
9  }
```

Comando `for` Aprimorado

O comando `for` aprimorado permite o acesso sequencial de leitura aos elementos de objeto contêiner, que pode ser um arranjo ou uma coleção. Uma coleção 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*, a qual não pode ser usada para modificar elementos do contêiner. Deve ser usada apenas para acesso a seus elementos.

O programa abaixo ilustra o uso do `for` aprimorado, calculando a soma dos elementos de um vetor de inteiros:

```
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 }
```

Rótulos

Comandos podem ser rolutados para que comandos `break` ou `continue` façam a eles referências. O rótulo é um identificador que precede um comando e somente pode haver um rótulo por comando:

rótulo `:` *comando*

Comando `continue`

Comando `continue` somente pode ocorrer dentro de comandos `for`, `while` ou `do-while`.

Dentro de `while` ou `do-while`, `continue` causa a expressão boolean de controle de iterações do comando de repetiçao ser novamente avaliada e testada para decidir quanto à execução da próxima iteração.

Dentro de `for`, `continue` causa a expressão de *avanço* ser avaliada e a *condição* do comando de repetição ser novamente testada para decidir quanto à execução da próxima iteração. O comando `continue` causa o avanço do comando de repetição para a próxima iteração.

O comando `continue` tem dois formatos:

- `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 indicado no comando.

O programa abaixo conta os números do intervalo $[3, 10.000.000]$ que não são primos. O comando de repetição mais interno é interrompido assim que um divisor for encontrado:

```
1  public class Continue1 {
2     public static void main(String[] args) {
3        int i = 3, j, x;
4        L: while (i < 10000000) {
5             j = 2;
6             while(j < i) {
7                if (i % j ==  0) {
8                    k++; i++;
9                    continue L;
10               }
11               j++;
12            }
13            i++;
14       }
15       System.out.print("Não-primos encontrados = " + k);
16    }
17 }
```

Comando `break`

O comando `break` somente pode ocorrer dentro de comandos `switch`, `for`, `while` ou `do-while`. Ele tem 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` envolvente que tem o rótulo indicado

O programa `Break1` também conta todos os números do intervalo $[3, 10.000.000]$ que não são primos. O comando de repetição mais interno é interrompido assim que um divisor for encontrado, evitando completar desnecessariamente 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) {
8                    k++;
9                    break L;
10               }
11               j++;
12            }
13            i++;
14       }
15       System.out.print("Não-primos encontrados = " + k);
16    }
17 }
```

O programa ilustra o uso de `break` e `break` com rótulo dentro de comando `switch`:

```
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` finalizam o comando `switch`. O resultado impresso pelo programa é `(0,21) - Valor final de a = 100`.

Comando `return`

O comando `return` tem o formato:

`return` *expressão* `;`

No caso de funções que retornam `void`, *expressão* deve se omitida.

Comando `throw`

O comando `throw` tem o formato:

`throw` *expressão* `;`

Comando `try`

A habilitação e tratamento de exceção em Java é obtida pelo comando `try`, que tem o formato:

`try` *bloco lista-de-catches*

Comando `Synchronized`

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

`synchronized (` *expressão* `)` *comando*

A *expressão* deve produzir uma referência para um objeto.

O comando

`synchronized x` *comando*

assegura que somente uma *thread* dentre todas que executarem o mesmo comando `synchronized` ou outro `synchronized y`, onde `y` refere-se ao mesmo objeto que `x`, adquire o direito de executar respectivo *comando* do comando `synchronized`.

1.10 Entrada e Saída Básicas

A bibioteca padrão de Java para administar arquivos de fluxo de entrada e saída é rica e sofisticada, contendo cerca de 64 classes. Contudo, reconhece-se que, para a implementação de programas que exerçam um mínimo de atração para um programador iniciante, é preciso tornar disponível o quanto antes recursos de fácil uso para realizar entrada e saída básicas de dados.

Neste sentido, antecipa-se a apresentação da classe `JOptionPane` da biblioteca `javax.wing`, a ser estudada em profundidade no Capítulo 12, 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. Esta biblioteca implementa algumas operações fundamentais de arquivos de fluxo, envolvendo o teclado, a tela do monitor de vídeo e arquivo sem formatação armazenado em memória secundária. As classes que compõem esta biblioteca são: `Kbd`, `InText`, `Screen` e `OutText`.

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. A classe possui um grande número de operações, dentre as quais destacam-se:

- `static int showConfirmDialog(null, Object m)`: exibe uma janela que pede uma resposta de confirmação `yes/no/cancel`.
- `static String showInputDialog(null, 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 das funções acima é do tipo `Component` e serve para indicar o componente da tela do monitor onde a janela de comunicação deve ser exibida. A passagem de um argumento com valor `null`, como indicado acima, faz a janela aparecer no centro da tela.

A execução do programa `Diálogo1` abaixo

```
1 import javax.swing.JOptionPane;
2 public class Diálogo1 {
3    public static void main( String args[] ) {
4       JOptionPane.showMessageDialog(
```

Figura 1.1 Mensagem Informativa de JOptionPane

.

```
5           null, "Welcome\nto\nJava\nProgramming!");
6        System.exit( 0 );
7     }
8  }
```

produz a mensagem informativa da Figura 1.1.

O programa `Adição` usa a operação `showInputDialog` para solicitar os valores de dois números inteiros e exibe de volta, por meio de `showMessageDialog`, o valor da soma destes dois inteiros.

```
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("Entre com primeiro inteiro");
6      s2=JOptionPane.showInputDialog("Entre com segundo inteiro");
7      sum = Integer.parseInt(s1) + Integer.parseInt(s2);
8      JOptionPane.showMessageDialog(
9         null, "The sum is " + sum, "Results",
10        JOptionPane.PLAIN_MESSAGE);
11     System.exit( 0 );
12   }
13 }
```

A execução do program produz, em sequência, exibindo uma de cada vez, as janelas mostradas na Figura 1.2.

Classe `Kbd`

Os métodos de `Kbd` tem nome segundo o padrão `read`*tipo*, onde *tipo* pode ser `Int`, `Long`, `Float`, `Double` ou `String`. Estes 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. A sequência lida é automaticamente convertida para `int`, `float`, `double` ou `String`, conforme cada um dos métodos de leitura:

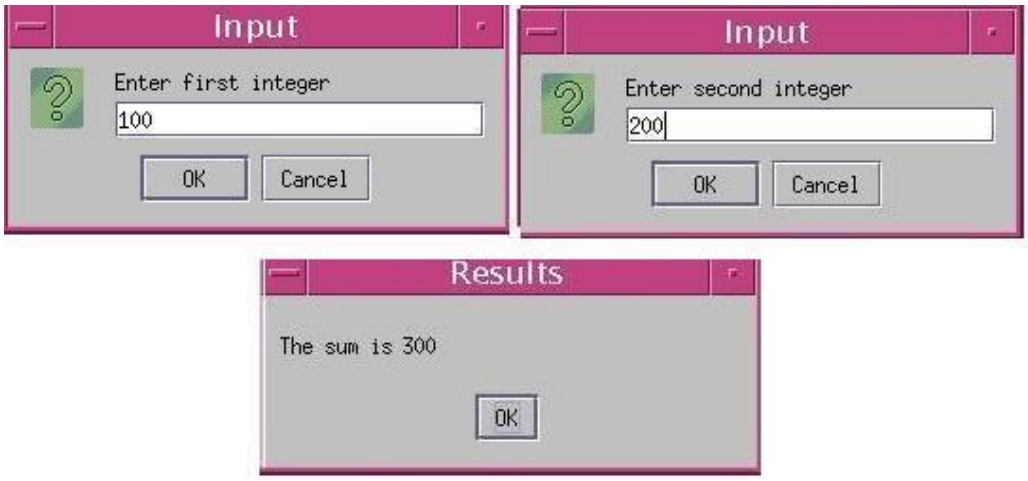


Figura 1.2 Interação com JOPtionPane

- `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`: retorna a sequência de caracteres que se inicia com um caractere diferente de branco e de caractere de controle e que termina com um branco ou um caractere de controle.

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();
```

Classe `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 toma as providências para sua abertura.

- `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`

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

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

Classe `Screen`

Os métodos da classe `Screen`, listados abaixo, convertem o parâmetro dado para uma sequência de caracteres ASCII e os 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.

- `final static public void println()`
- `final static public void print(String s)`
- `final static public void println(String s)`
- `final static public void print(boolean b)`
- `final static public void println(boolean b)`
- `final static public void print(byte b)`
- `final static public void println(byte b)`
- `final static public void print(char c)`
- `final static public void println(char c)`
- `final static public void print(int i)`
- `final static public void println(int i)`
- `final static public void print(long n)`
- `final static public void println(long n)`
- `final static public void print(float f)`
- `final static public void println(float f)`
- `final static public void print(double d)`
- `final static public void println(double d)`

O trecho de programa a seguir mostra a exibição na tela do monitor de uma sequência de caracteres informando o valor do inteiro `i`:

```
int i = 10;
Screen.print("Valor de i = ");
Screen.println(i);
```

O programa `TestKS` abaixo é um exemplo completo, que mostra o uso das classes `Kbd` e `Screen`:

```
1  public class TestKS {
2    public static void main(String[] args) throws IOException {
3       boolean b = false;  int x = 0;  long r = 0;
4       float y =(float) 0.0;  double d = 0.0;
5       String s = "indefinido";
6       Screen.println("Entre com os valores:");
7       Screen.print("boolean b = ");
8       b = Kbd.readBoolean() ;
9       Screen.print("b lido = "); Screen.println(b);
10      Screen.print("int x  = ");
11      x = Kbd.readInt() + 1;
12      Screen.print("x + 1 = "); Screen.println(x);
```

```
13        Screen.print("long r  = ");
14        r = Kbd.readLong() + 1 ;
15        Screen.print("r + 1 = "); Screen.println(r);
16        Screen.print("float y = ");
17        y = Kbd.readFloat() + (float) 1.0;
18        Screen.print("y + 1.0 = " ); Screen.println(y);
19        Screen.print("double d = ");
20        d = Kbd.readDouble() + 1.0;
21        Screen.print("d + 1.0 = "); Screen.println(d);
22        Screen.print("String s = ");
23        s = Kbd.readString();
24        Screen.print("s = "); Screen.println(s);
25     }
26  }
```

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.

- `public OutText(String fileName) throws IOException`
- `final public void println()`
- `final public void print(String s)`
- `final public void println(String s)`
- `final public void print(boolean b)`
- `final public void println(boolean b)`
- `final public void print(byte i)`
- `final public void println(byte i)`
- `final public void print(short i)`
- `final public void print(String s)`
- `final public void println(short i)`
- `final public void print(int i)`
- `final public void println(int i)`
- `final public void print(long n)`
- `final public void println(long n)`
- `final public void print(float f)`
- `final public void println(float f)`
- `final public void print(double d)`
- `final public void println(double d)`
- `final public void println(String s)`

O trecho de programa abaixo escreve no arquivo `C:\MeusArquivos\saída.txt` a sequência de caracteres `"\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();
o.println(i);
o.print(k);
```

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

Na prática, o erro mais provável é a leitura indevida da marca de fim de arquivo. Para evitar o cancelamento do programa por este 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 programar o tratamento da exceção `IOException`, conforme mostra o programa abaixo, embora outros tipos de erro possam ser mascarados pela mensagem emitida.

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

1.11 Ambientes e Escopo de Nomes

Um identificador serve 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. Para isto, há em Java dois mecanismos. O primeiro, 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 cinco ambientes 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 desnecessária. Ou ainda melhor, usar repetições para melhorar a legibilidade do programa. Embora permitido, o uso de um mesmo identificador para designar muitos elementos distintos, como mostrado no programa abaixo, não é recomendável:

```
package X;
class X {
   static X X = new X( );
   X X(X X) {
       X: while (X == X.X(X)) {
           if (X.X.X != this.X)  break X;
       }
       return new X( );
   }
}
```

O segundo mecanismo para identificação de nomes é a regra de escopo das declarações. Declarações de nomes não-públicos em um pacote têm visibilidade restringida ao pacote. O escopo de uma variável de método é o bloco onde foi declarada. O de parâmetros é o corpo de seu método.

1.12 Exercícios

1. Por que a ordem de avaliação dos operandos pode não corresponder à ordem de aplicação dos operadores em uma expressão?

2. Há linguagens, como o Pascal, que são orientadas a comandos. Outras, como Algo68, que são orientadas a expressões. Estude estes conceitos e determine como Java poderia ser classificada.

3. Por que efeito colateral em expressões pode ser nocivo à legibilidade do programa?

4. Explique o significado de **semântica de valor** e de **semântica de referência**.

5. 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 }
```

1.13 Notas Bibliográficas

As mais importantes referências para a linguagem Java são as páginas da Sun Microsystem Inc (http://www.javasoft.sun.com) e os livros publicados pelos criadores da linguagem Java [1, 2].

O livro dos Deitel [8] é uma boa referência para aqueles que veem Java pela primeira vez. Embora seja um pouco verboso, este 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 estes objetos de forma que objetos de computação representem objetos da vida real.

Objetos manipulados por um programa são abstrações de dados, implementadas por meio de uma estrutura de dados e um conjunto de operações. Assim, objetos têm uma estrutura interna, definida por uma estrutura de dados, e seus estados são representados 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 respeito à representação dos objetos e sua manipulação exclusivamente via as operações definidas.

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 objetos. Classe é um recurso linguístico para descrever o comportamento de um conjunto de objetos. Classe é, desta forma, um molde para construir objetos. Para isto, o conceito de classe oferece 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 entender seu comportamento. Associado a classes há os conceitos de **Abstração**, **Encapsulação**, **Proteção de Dados**, **Ligação Dinâmica Herança**.

Abstração trata dos mecanismos que permite que se concentre nas partes relevantes e ignorar o que não for relevante naquele 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 fora das operações definidas para a classe.

Ligação dinâmica de nomes a funções 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.

Em geral, a declaração de uma classe define um novo tipo abstrato de dados, embora outras entidades também possam ser definidas por meio de classes.

Os principais elementos constituintes de uma classe são: membros de dados, também chamados de campos ou atributos, funções construtoras , função finalizadora , funções ou métodos e blocos de iniciação .

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. Este relacionamento é também chamado de **tem-um**.

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

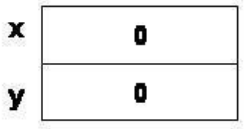
A função finalizadora é uma função especial que automaticamente é invocada no momento em que a área de um objeto estiver sendo recolhida pelo coletor de lixo.

As operações normais da classe são definidas por membros função.

A classe `PontoA` abaixo define um tipo abstrato de dados de nome `PontoA`, caracterizado pelas operações `set`, `getX`, `getY` e `clear`. Os campos `x` e `y` foram declarados de acesso privativo às operações definidas na classe, e as funções de nome `PontoA` são funções construtoras.

```
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 void print(" (" + this.x + "," + this.y + ")");
10 }
```

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 figura abaixo:



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`. O

endereço do objeto sobre o qual as operações se aplicam em cada invocação é identificado pela referência implicitamente definida `this`, usada explicitamente nos corpos de métodos da classe `PontoA`.

Por economia de escrita, a referência `this`, em muitas situações, pode ser omitida, como mostra a seguinte declaração de uma classe `PontoB` equivalente à classe `PontoA`:

```
1 public class PontoB {
2    private int x, y;
3    public PontoB(int a, int b) {
4       x = a; y = b;
5    }
6    public void clear( ) {
7       x = 0; y = 0;
8    }
9 }
```

As ocorrências de `x` e `y` nas linhas `PontoB.4` e `PontoB.7` devem ser entendidas como `this.x` e `this.y`, respectivamente.

2.1 Criação de Objetos

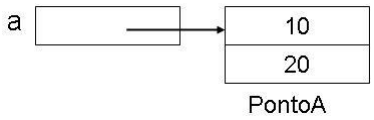
A área de dados de um programa normalmente consiste em duas partes: **Pilha de Execução** e **Heap**. As variáveis locais a métodos são alocadas na pilha de execução, automaticamente no início da execução do método que as contém, e liberadas no ato de retorno do método. O mesmo vale para os parâmetros 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 seguinte estrutura para estas declarações:

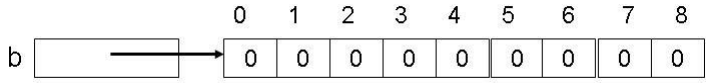
A criação de objetos a partir de uma classe `A` dá-se pela operação `new A(args)`, que realiza as seguintes ações:

- 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;
- 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 do objeto `PontoA` criado, como mostra a figura:



A criação de objetos do tipo arranjo é obtida pela operação `new A[t]`, onde `t` é uma expressão que dá o tamanho do arranjo a ser alocado e `A` o tipo de seus elementos. Esta operação aloca o arranjo no tamanho indicado, inicia seus campos com valores *default* e retorna o endereço da área alocada. 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 o seguinte resultado:



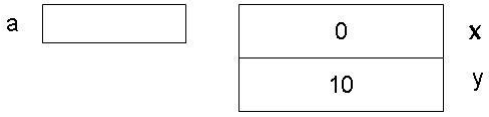
Para ver o processo de criação de objetos em detalhes, considere o programa `Criação` abaixo:

```
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 }
```

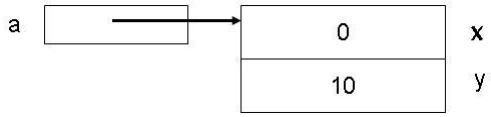
A execução deste programa inicia-se na linha `Criação`.3 com o processamento da declaração `M a`, que causa a alocação, na pilha de execução, da referência `a`, com valor indefinido, produzindo:

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.

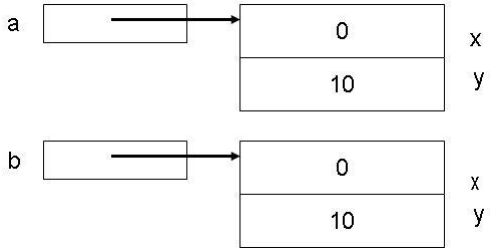
A seguir, ainda na linha `Criação`.3, a operação `new M()` cria um objeto do tipo `M` na área do *heap*, iniciando o campo `x` com o valor default `0`, e o campo `y`, com o valor indicado em sua declaração:



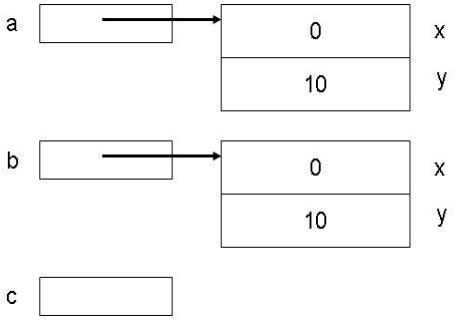
A execução do comando da linha `Criação`.3 conclui-se com a atribuição do endereço do objeto alocado e inicializado à referência `a`, produzindo o seguinte mapeamento da memória até este ponto:



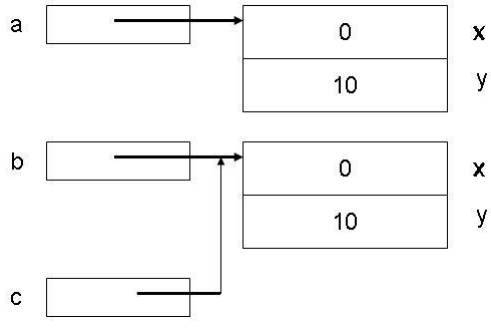
A execução da linha `Criação`.4 causa a alocação de um novo objeto, também do tipo `M`, e armazena seu endereço na referência declarada `b`, conforme mostra-se:



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

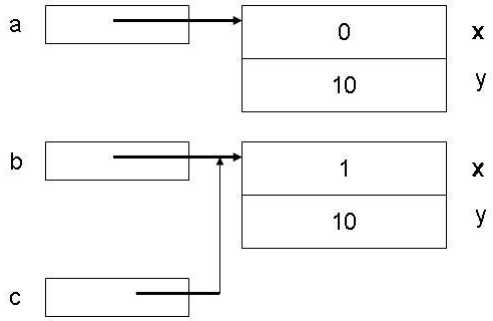


O comando de atribuição da linha `Criação`.6 cria dois caminhos de acesso a um mesmo objeto, via as referências `b` e `c`, criando um compartilhamento. Note que a atribuição é de referência e não de objeto:

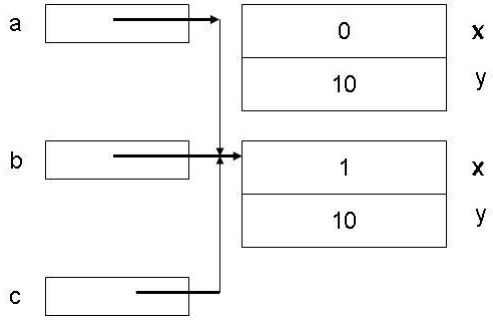


Para se fazer cópias de objetos, deve-se usar o método `clone` definido na classe `Object`, conforme detalhado no Capítulo 14.

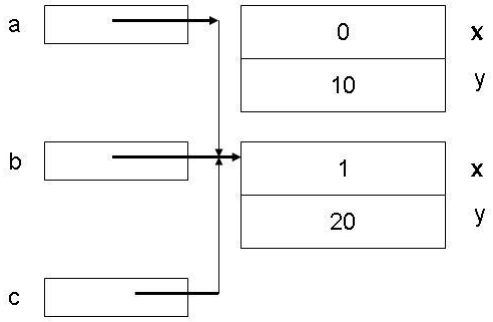
A linha `Criação`.7 mostra que há dois caminhos de acesso aos campos do objeto apontado por `b` e `c`. O resultado, após executar o comando de atribuição desta linha, é:



Na linha `Criação`.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 `Criação`.9, é:



2.2 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 e parâmetros de métodos. Estes elementos têm visibilidade limitada ao corpo do método. Em segundo nível, tem-se a visibilidade dos membros de classes, que normalmente pode estar restrita à classe ou ao pacote.

A definição de pacote de Java pode ser encontrada no Capítulo 8. No momento é suficiente saber que um pacote é uma coleção de classes que detêm algum privilégio de acesso entre si, e que um pacote estabelece 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.

Os principais constituintes de uma classe sujeitos ao controle de visibibidade têm uma das formas:

- campos: *modificador-de-acesso tipo nome-do-campo*;
- métodos: *modificador-de-acesso tipo nome-do-método(parâmetros) corpo*;
- construtores: *modificador-de-acesso nome-da-classe(parâmetros) corpo*;

Onde o *modificador-de-acesso* denota o nível de visibilidade do elemento declarado. As opções são:

- `public`: membros declarados como `public` são acessíveis de qualquer lugar onde a classe for acessível, inclusive nas respectivas subclasses.
- `private`: membros declarados como `private` somente são acessíveis aos membros funcionais 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.
- *vazio*: membros declarados sem modificador de acesso são acessíveis somente por classes que estiverem dentro do mesmo pacote.

Para ilustar o uso de modificadores de acesso, considere a classe `Ponto` definida, para fins de exemplo, com todos os seus membros declarados públicos:

```
1  public class Ponto {
2     public double x, y;
3     public void clear( ) { x = 0; y = 0; }
4  }
```

Suponha que no pacote onde se encontra a classe `Ponto`, está também declarada a classe `TrincaDePontos`, que possui, a título de exemplo, campos com visibilidade privada, protegida e pacote:

```
1  public class TrincaDePontos {
2     private Ponto p1;
3     protected Ponto p2;
4     Ponto p3;
5     public TrincaDePontos( ) {
6        p1 = new Ponto( );
7        p2 = new Ponto( );
8        p3 = new Ponto( );
9     }
10    public void clear( ) {
11       p1.clear();
12       p2.clear();
13       p3.clear();
14    }
15 }
```

Uma terceira classe, de nome `Visibilidade1`, também declarada no mesmo pacote que contém `Ponto` e `TrincaDePontos`, tem acesso aos elementos destas classes, excetuando-se o campo `p1`, que é declarado `private`. O erro indicado na linha `Visibilidade1.7` deve-se a este fato.

```
1  public class Visibilidade1 {
2     public static void main(String[ ] args) {
3        Ponto a = new Ponto( );
4        TrincaDePontos p = new TrincaDePontos( );
5        Ponto b, c, d;
6        a.x = a.y + 1; // OK
7        b = p.p1;      // ERRADO
8        c = p.p2;      // OK
9        d = p.p3;      // OK
10       p.clear();     // OK
11    }
12 }
```

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

```
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 }
```

2.3 Métodos

Métodos de Java somente podem ser declarados no corpo de classes e correspondem ao conceito das tradicionais funções em linguagens de programação. Métodos podem ou não retornar valores. Métodos que não retornam valores são chamados de funções `void`, pois este é o tipo que deve ser declarado para o seu valor de retorno.

Os parâmetros dos métodos são alocados na pilha de execução no momento da entrada do método e liberadas no seu retorno. Os parâmetros, que são iniciados pelos argumentos de chamada, são sempre passados **por valor**. Isto significa que 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 uma referência de um objeto como parâmetro, somente o valor da referência é efetivamente copiado para a área de execução do método. Objetos nunca são copiados, sejam eles arranjos ou um objeto qualquer. Não há passagem de parâmetros por referência em Java, mas referências podem ser passadas por valor.

Desta forma, estando dentro de um método, é possível, via uma referência dada por um parâmetro, modificar os objetos externos ao método. Entretanto, o parâmetro de chamada em si é sempre inacessível para modificações, conforme ilustram os programas `Passagem1` e `Passagem2`.

```
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 acima produz a saída: `A: x = 1 M: y = 2 D: x = 1`

O programa `Passagem2`

```
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();
9        altera2(y);
10       for (int z: y) System.out.print(" " + z);
11    }
12 }
```

que usa a classe `PontoA` definida na Seção 2, produz a saída: `(10,10) (0,0) 1 2 4`

A operação de chamar uma das funções de uma classe para um dado objeto, por exemplo, `p.clear()`, é interpretada como **enviar a mensagem `clear` ao objeto receptor `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.

2.4 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 objetos são iniciados com o valor `null`. Ainda durante a execução do operador `new`, as iniciações indicadas nas declarações dos campos são processadas, e por último o corpo de uma das funções construtoras declaradas na classe, selecionadas conforme os tipos dos parâmetros, é executada.

Uma função construtora distingue-se dos demais métodos definidos em 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. Muitas classes não têm função construtora explicitamente especificada. 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 os tipos desses parâmetros.

Na prática, os corpos das funções construtoras de uma mesma classe quardam 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**. A palavra-chave `this`, em Java, serve a dois propósitos: (i) a palavra-chave `this` serve para denotar o endereço do objeto receptor de uma mensagem; (ii) representa a função construtora identificada por 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     ...
6  }
```

A classe 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.

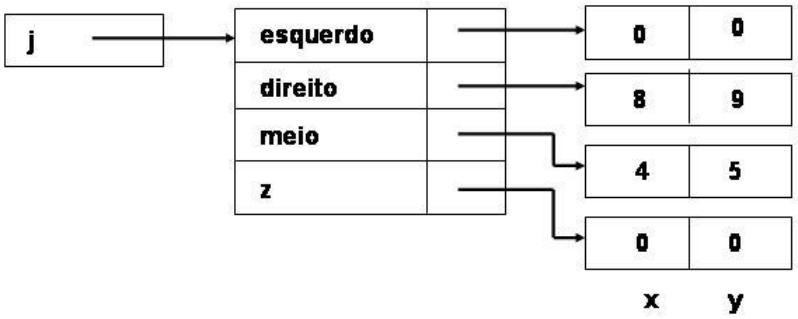
```
1  public class TesteDeConstrutora {
2     Tipo1 a;
3     Tipo2 b;
4     Construtora1 x = new Construtora1( );
5     Construtora1 y = new Construtora1(a);
6     Construtora1 z = new Construtora1(a,b);
7  }
```

O programa `Janela` 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( )` produz:



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` impede que objetos da classe sejam criados fora da classe. E se a função construtora for `protected`, somente a própria classe ou seus herdeiros poderão criar objetos da classe.

Se não houver, na classe, definição alguma de construtoras, por *default* é criada uma sem parâmetros. Ressalva-se que construtora *default* somente é criada neste caso.

2.5 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 liberado, de forma que a área que ocupa na memória *heap* pode ser liberada para ser reusada.

O sistema de coleta de lixo da JVM automaticamente administra a reciclagem 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 o objeto que os controla encerrar 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 o 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 esta dificuldade, Java possibilita a definição de uma função membro especial, de nome `finalize`, que é executada automaticamente quando a área do objeto da classe que a contém estiver para ser liberada pelo coletor de lixo. Por exemplo, na classe `ProcessaArquivo`, 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) { arquivo.close(); arquivo.null(); }
8     }
9     protected void finalize() throws Throwable {
10       super.finalize(); fechaArquivo();
11    }
12 }
```

O método `finalize` é um método `protected` definido na classe `Object`, e que pode ser redefinido pelo programador em sua classe. Toda classe Java é uma extensão direta ou indireta de `Object`, cujos métodos podem ser redefinidos em subclasses ou classes estendidas, 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, por exemplo, o significado de `super`, que aparece na linha `ProcessaArquivo`.10, podem ser encontrados no Capítulo 3.

Há o real perigo de a operação `finalize` ressuscitar seu objeto corrente, criando 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 única vez para cada objeto. Consequentemente objetos ressuscitados 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 simplesmente *ressuscitá-lo*.

Como não se tem garantia de que o coletor de lixo será ou não ativado durante a execução do programa, a coleta do objeto sem uso pode nunca ocorrer e, consequentemente, não se pode garantir que a função `finalize` do objeto seja sempre executada. Para ilustrar este fenômeno, observe os valores impressos pelos programas `G1` e `G2` abaixo, nos quais, dois objetos são alocados, sendo um deles explicitamente liberado ao se atribuir um novo valor à sua referência. As funções `finalize` definidas nesses dois 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 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  }
```

`Saída: Finalize foi chamada. Acabou!`

Para se assegurar que o coletor de lixo seja de fato chamado, o programa `G1` o ativa explicitamente via comando `System.gc()`, na linha `G1`.4, 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 `G1`.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 e nem seus serviços se fizeram necessários, dado a pequena demanda de memória do exemplo. Portanto, desta vez, a função `finalize` não foi executada como mostra a saída do programa.

```
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  }
```

`Saída: Acabou!`

Programa `G3` mostra que a chamada explícita 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 }
```

`Saída: Finalize foi chamada.`

Conclui-se que a função `finalize` é um recurso de programação importante, mas indevidamente 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 ser evitada 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.

2.6 Referência `this`

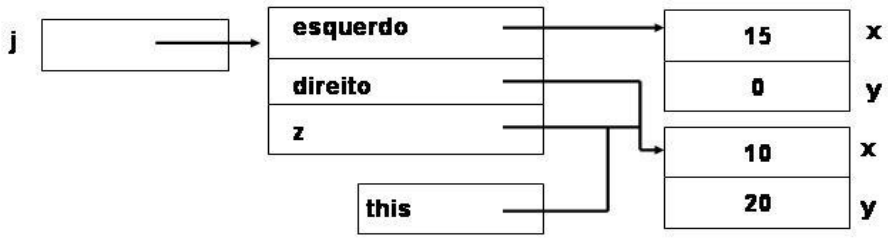
Um dos usos da palavra-chave `this` é prover automaticamente o endereço ou referência do objeto corrente durante a execução de um método. Por exemplo, a execução do seguinte trecho de código, que se refere a classe `Janela` definida a seguir,

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

indica que, durante a execução do método `set`, definido na linha `Janela`.5, a variável `this`, válida no escopo de `set`, tem o mesmo valor armazenado em `j`, o qual é o endereço do objeto corrente. Assim, a chamada ao método `clear`, na linha `Janela`.10 dentro do método `set` do programa abaixo, tem como objeto receptor o `Ponto` apontado pelo campo `z` de objeto referenciado por `j`.

```
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;
7        this.z = this.direito;
8        this.direito.x  = 10;
9        this.direito.y  = 20;
10       this.z.clear();
11    };
12 }
13 class Ponto {
14    public double x, y;
15    public void clear( ) {
16       this.x = 0; this.y = 0;
17    }
18 }
```

No início do método `clear` da linha `Ponto.??` tem-se a seguinte alocação de objetos:



Para se 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. Nesta 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 `Ponto.15` 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`.

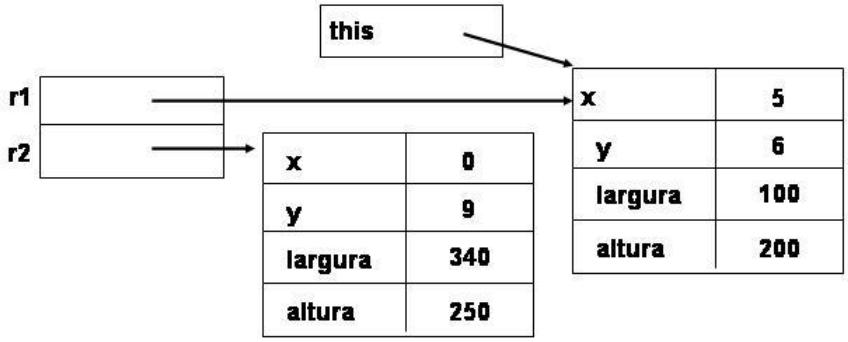
A referência `this` pode ser omitida sempre que puder ser deduzida pelo contexto, mas deve ser usada quando for necessário para explicitar as referências desejadas, como mostra o exemplo a seguir:

```
1  public class Rectangle {
2     private int x, y;
3     private int largura;
4     private int altura;
5     public Rectangle(int x, int y, int w, int h) {
6        this.x = x;
7        this.y = y;
8        this.largura = w;
9        altura = h;
10    }
11    public void identify() {
12       System.out.print("Retângulo (" + x + ","+ y + ") - ");
13       System.out.println("Dimensões: " + largura + " X " + altura);
14    }
15    public static void main(String args[ ]) {
16       Rectangle r1 = new Rectangle(5,5,100,200);
17       Rectangle r2 = new Rectangle(0,9,340,250);
18       r1.identify();
19       r2.identify();
20    }
21 }
```

Observe que o uso de `this` nas linhas `Rectangle.6` e `Rectangle.7` 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 `Rectangle.8` poderia ter sido omitido, como o foi na linha `Rectangle.9`, onde escreveu-se `altura` no lugar de `this.altura`.

A figura abaixo mostra a alocação dos objetos do programa `Rectangle` quando o fluxo de execução atingir o início do método `identify`, na linha `Rectangle.11`, após sua

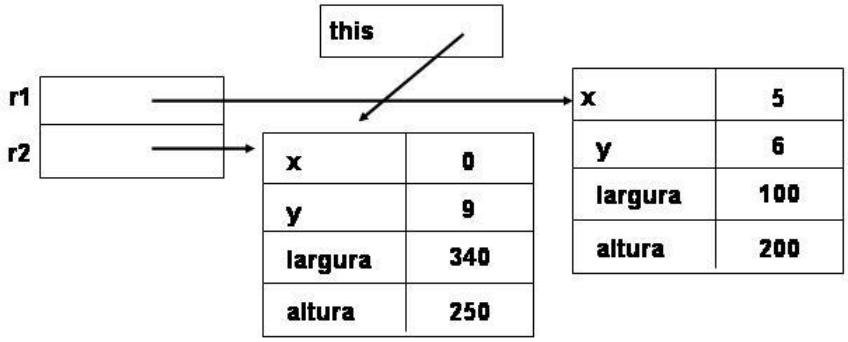
chamada pelo comando `r1.identify()` da linha `Rectangle`.18. Note que a referência `this` aponta para o mesmo objeto correntemente referenciado por `r1`:



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

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

Analogamente para a chamada `r2.identify()` tem-se a seguinte configuração:



O valor impresso pela execução de `r1.identify()` é:

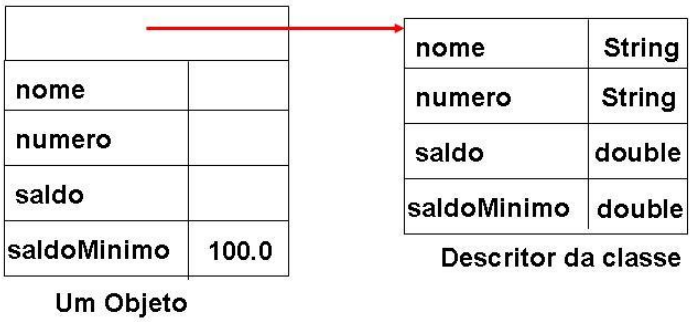
```
Retângulo (0,9) - Dimensões 340 X 250
```

2.7 Variáveis de Classe e de Objeto

Os atributos declarados em uma classe podem ser **variáveis de classe** ou **variáveis de instância** . As variáveis de instância são campos de cada objeto alocado, enquanto que as variáveis de classe têm instância única compartilhada por todos os objetos da classe. Para ilustrar as diferenças e aplicação destes dois tipos de variáveis, considere a classe `Conta1` definida por:

```
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 de sua classe:

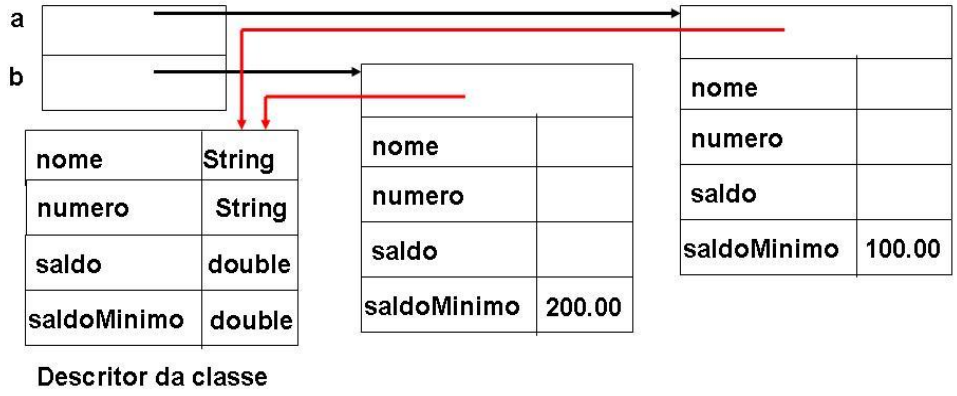


O *Descritor da Classe* que aparece na figura é um objeto do tipo `Class`, que descreve a estrutura da classe `Conta1`. Há um objeto do tipo `Class` para cada classe usada em um programa. Este objeto é alocado no momento em que o nome da classe for encontrado durante a execução, por exemplo, na declaração de uma variável ou de um parâmetro. Todo objeto possui um apontador para o descritor de sua classe como mostra a figura acima. Isto permite que se identifique o tipo de qualquer objeto durante a execução.

A aplicação `Banco1`, que utiliza a classe `Conta1`, definida por

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

A criação de um objeto pela operação `new Banco1()` produz a seguinte alocação de objetos:



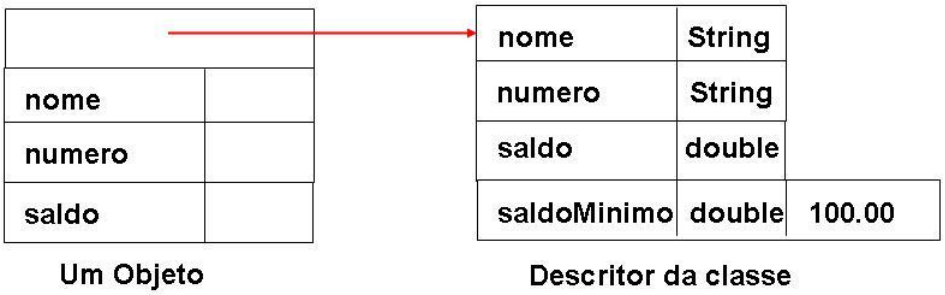
Note que todo objeto do tipo `Conta1` tem seu próprio saldo mínimo. Situações em que os valores de saldo mínimo variam de conta para conta, esta solução justifica-se. Entretanto, se o saldo mínimo tiver que ser o mesmo para todas as contas, seria um desperdício ter seu valor armazenado em cada uma das 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 este 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:

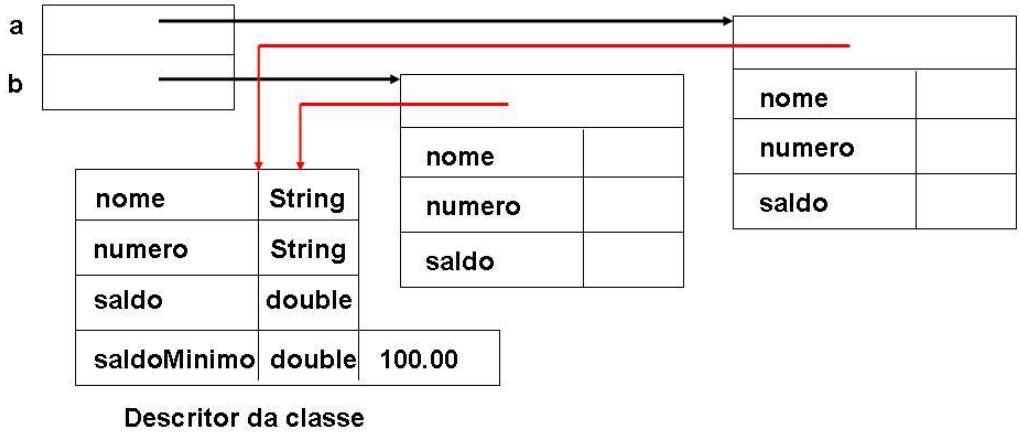


Note que o campo estático ou variável de classe `saldoMinimo` é alocado no próprio descritor da classe e não na área dos objetos que forem instanciados. Campos estáticos são automaticamente iniciados com valores *default*, na primeira vez em que a classe que os contém for encontrada durante a execução,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

No programa `Banco2`, duas contas são criadas com um mesmo saldo mínimo:

```
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:



Embora variáveis ou campos estáticos não pertençam ao leiaute dos objetos, eles podem ser tratados como se fossem, podendo ser qualificados pela referência `this`. 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:

```
a.saldoMinimo += 1000.00;
b.saldoMinimo  = 2000.00;
Conta2.saldoMinimo = 1000.00;
```

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` no exemplo abaixo:

```
1  class A {
2     static int x = 100;
3     static private int f( ) {
4        return x;
5     }
6  }
7  public class TestedeA {
8     public static void main(String[] args) {
9        System.out.println("f = " + A.f( ));
10    }
11 }
```

Métodos estáticos não possuem o parâmetro implícito `this`, que foi discutido na Seção 2.6. 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`:

```
1  class B {
2     static int x = 100;
3     static private int f() {
4        return this.x;
5     }
6  }
7  public class TestedeB {
8    public static void main(String[] args) {
9       System.out.println("f = " + B.f());
10   }
11 }
```

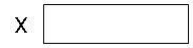
2.8 Arranjos de Objetos

Arranjos de objetos são de fato arranjos de referências a objetos. Quando um arranjo deste 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 este processo, considere o programa abaixo, que contém um arranjo de tipo primitivo `int` e um outro de referências a objetos:

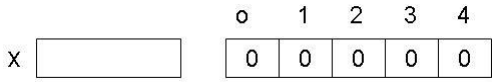
```
1  public class Arranjo {
2     public static void main(String[] a) {
3        int[ ] x = new int[5];
4        x[2] = x[1] + 1;
5        A [ ] y = new A[5];
6        y[2] = new A(1,2);
7        y[5] = new A(4,6);
8     }
```

```
9  }
10 class A {
11     private int r, s;
12     public A(int d, int e) { r = d; s = e; }
13 }
```

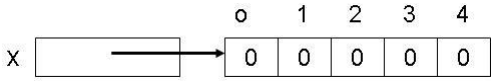
O efeito da declaração `int[ ] x`, da linha `Arranjo`.3, é a alocação, na pilha de execução da JVM, de uma célula não iniciada `x`:



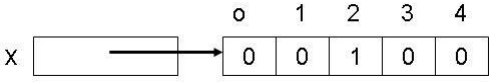
A operação `new int[5]`, da linha `Arranjo`.3, aloca um arranjo de 5 inteiros e devolve seu endereço:



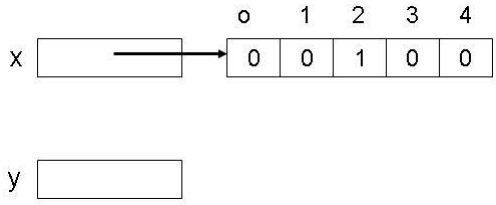
A atribuição da linha `Arranjo`.3 inicia `x` com o endereço do arranjo criado:



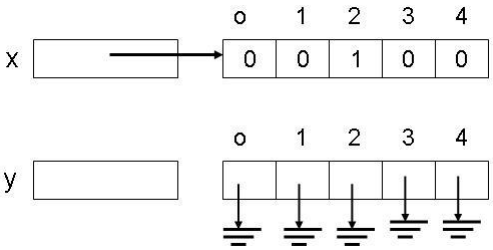
O comando de atribuição `x[2] = x[1] + 1`, da linha `Arranjo`.4, atualiza o elemento de arranjo `x[2]` com o valor `1`:



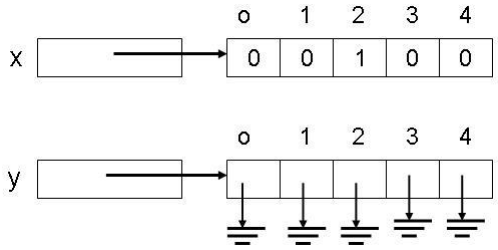
A declaração `A[ ] y`, da linha `Arranjo`.5, aloca `y` na pilha de execução, produzindo:



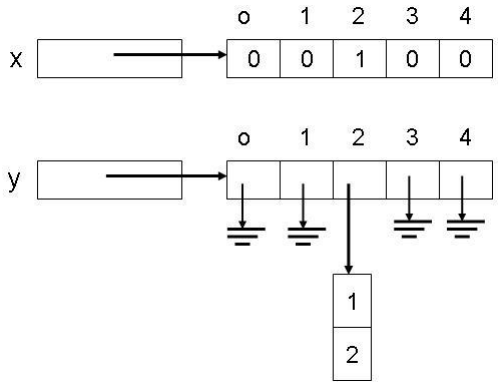
A operação `new A[5]`, da linha `Arranjo`.5, aloca no **heap** um arranjo de cinco referências para objetos do tipo `A`. Os elementos deste arranjo são iniciados como o valor `null`:



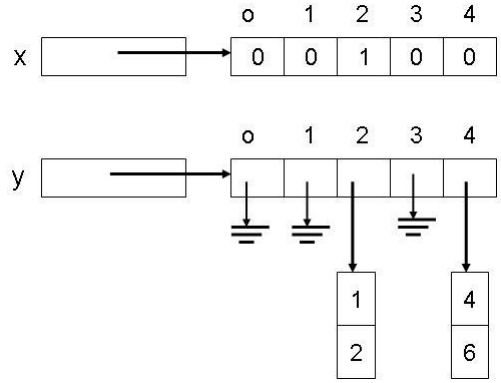
Após a execução do comando de atribuição da linha `Arranjo`.5, tem-se a seguinte configuração de objetos:



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



Após a execução do último comando, o da linha `Arranjo`.7, a configuração final dos objetos criados pelo programa `Arranjo` é:



2.9 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;
- 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 for criado. A iniciação destes campos ocorre segundos 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, todos serão executados em ordem de sua ocorrência, dentro de sua categoria, estáticos ou não-estáticos.

O programa abaixo 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 `IniciaNãoEstáticos`.2, há um iniciador de declaração, na linha `IniciaNãoEstáticos`.3, está um bloco de iniciação de não-estáticos e, nas linhas `IniciaNãoEstáticos`.4 e `IniciaNãoEstáticos`.7, estão duas funções construtoras.

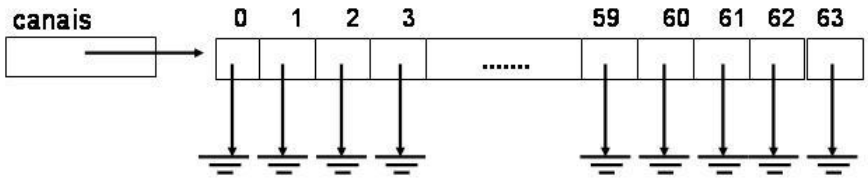
O objetivo do exemplo acima é mostrar os passos da execução de iniciação de objetos, que produz saída: `x1 = 110 x2 = 1110 x1 = 110 x4 = 10110`, cuja análise revela o fluxo de execução, mostrado, a seguir, linha a linha.

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

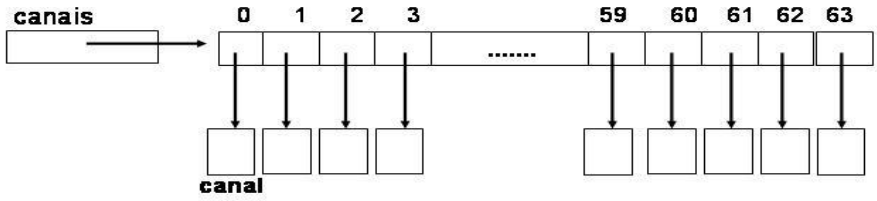
O exemplo a seguir exibe a necessidade de se usar blocos de iniciação de variáveis estáticas, porque as operações de iniciação necessárias são mais complexas:

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



A seguir, o bloco de iniciação de estáticos contido nas linhas `IniciaEstáticos`.5 a `IniciaEstáticos`.9 produz:



completando a iniciação do variável de classe `canais`.

2.10 Alocação de Variáveis de Classe

Variáveis de instância são alocadas e iniciadas no momento da criação do objeto. Entretanto, a alocação e iniciação de variáveis de classe é mais complexa. O exemplo a seguir mostra quando estes eventos ocorrem em um programa Java. As classes `A`, `B`, `C` e `D` têm toda a mesma estrutura: declaração e iniciação direta de dois campos estáticos, um bloco de iniciação de estáticos, um bloco de iniciação de não-estáticos e uma função construtora.

```
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 }
25 class X {
26   public static int f(char m){System.out.print(" " + m);return 1;}
27 }
```

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

```
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();} else {D g = new D();};
13        o.println("Acabou");
14     }
15  }
```

O programa `LoadClass5` inicia sua execução com a criação de um objeto do tipo `A` na linha `LoadClass5`.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
```

Na linha `LoadClass5`.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 `LoadClass5`.7.

Observe que na linha `LoadClass5`.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 `LoadClass5`.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 `LoadClass5`.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 `LoadClass5`.10, a classe `B` já está alocada, e, portanto, também já estão seus campos estáticos. Assim, neste 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
```

Como o programa foi executada sem parâmetros na linha de comando, portanto `args.length` tem valor `0`, a saída abaixo, que foi produzida pela linha `LoadClass5`.12, mostrando que nada da classe `D` foi alocado.

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

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
```

2.11 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.

A conta de poupança é modelada pela seguinte classe `Conta`, 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 a previsão para, no caso de haver mais de uma conta, todas serem remuneradas pela mesma taxa de juros foi modelada por meio do membro `taxaAnual`, linha `Conta`.2, que foi declarado como uma variável de classe.

Para exibir a evolução, o programa `Poupança` lê da linha de comando valor do depósito mensal e a taxa anual de juros:

```
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);
16                c.creditar(deposito);
17                System.out.print(c.saldo() + ", ");
```

```
18              }
19              System.out.println();
20          }
21  }
```

2.12 Exercícios

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

2. Quando uma classe define um tipo abstrato de dados?

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

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

2.13 Notas Bibliográficas

As mais importantes referências para a linguagem Java são as páginas da Sun Microsystem Inc (http://www.javasoft.sun.com) e os livros publicados pelos criadores da linguagem Java [1, 2].

O livro dos Deitel [8] é uma boa referência para aqueles que veêm Java pela primeira vez. Embora seja um pouco verboso, este livro apresenta muitos de exemplos esclarecedores do funcionamento da linguagem Java.

Capítulo 3

Hierarquia

Tipos de dados classificam os dados, provendo uma organização nos valores usados em um programa. A separação dos dados em tipos, como inteiros, fracionários, cadeias de caracteres e booleanos, permitem implementação eficiente das operações de cada um destes tipos, além de facilitar a detecção de erros de mistura indevida de tipos.

Tipos (ou classes) por sua vez também podem ser organizados de forma a constituírem famílias que compartilham propriedades e operações. A partir dos conjuntos das operações comuns a dois ou mais tipos pode-se formar, recursivamente, uma hierarquia de tipos, onde cada elemento descreve as operações comuns a ele e a todos os seus descendentes. Assim, objetos cujo tipo participa de uma hierarquia podem ser usados onde for esperado um objeto de tipo igual ou superior ao deles nesta 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-se uma classe existente e dela herdando declarações de atributos e implementação de métodos. Neste processo de reúso, pode-se, dentro de certos limites, adaptar os elementos herdados e, até mesmo, 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** desse 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.

3.1 Interfaces

Interface de Java é um recurso linguístico para especificar as assinaturas das operações de um tipo abstrato de dados a ser implementado por meio da declaraçao de uma classe. Uma interface consiste na encapsulação de declarações de cabeçalhos de métodos e de constantes simbólicas, estas declaradas como campos estáticos. A visibilidade de qualquer membro de interface é sempre `public`.

Em uma interface, métodos não podem ter seu corpo definido e também não podem ser finais e nem estáticos. Além disto, métodos de interface não podem ter modificadores

de características de implementação, como `native`, `synchronized` ou `strictfp`. Interfaces valorizam a independência de implementação.

Não se pode criar objetos a partir de sua interface, uma vez que somente os cabeçalhos dos métodos são definidos e a representação dos objetos nunca está definida na interface. Objetos devem ser criados a partir de classes concretas que implementem a interface. Estas classes concretas devem declarar os membros de dados que forem necessários e dar corpo aos métodos da interface. Mais de uma classe pode implementar uma mesma interface e objetos destas classes têm o tipo definido pela interface.

Para ilustrar as vantagens do uso de interfaces no lugar de classes concretas, considere as seguintes declarações:

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

A classe `Usuário1`, abaixo, possui dois métodos que fazem exatamente a mesma tarefa, diferindo apenas nos tipos dos objetos que aceitam como parâmetros. Isto exibe uma situação de baixa reusabilidade de código, manifestada pela duplicação dos corpos dos métodos `g1` e `g2`.

```
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 teste1( ){
5        Bolo   b = new Bolo(150.0, 19.0);
6        Torta  t = new Torta(300.0, 56.0);
7        double z = g1(b) + g2(t);
8        System.out.println(b.identidade() + ":");
9        System.out.println("  " + b.calorias() + " kcal");
10       System.out.println("  " + b.gorduras() + "g de gorduras");
11       System.out.println(t.identidade() + ":");
12       System.out.println("  " + t.calorias() + " kcal");
13       System.out.println("  " + t.gorduras() + "g de gorduras");
14       System.out.println("Total de Calorias de = " + z);
15    }
16 }
```

Duplicação de código é 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 da duplicação desnecessária de código e aumenta o grau de reúso.

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 classe `Usuário1` como tendo exatamente as mesmas operações — `identidade()`, `calorias()` e `gorduras()`. Isto sugere colocar os tipos dos objetos `Bolo` e `Torta` na mesma família definida por uma interface comum e, assim, prover um uso uniforme dos objetos `Bolo` e `Torta`.

As operações comum das classes `Bolo` e `Torta` podem então ser reunidas na interface:

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

Esta interface representa a especificação das operações de um tipo abstrato de dados, que pode ser implementado por meio de classes, como `Bolo` ou `Comestível`, que são duas implementações independentes do mesmo tipo `Comestível` e que iniciam a formação de uma hierarquia de tipos centrada em `Comestível`. Figura 3.1 exibe a hierarquia formada pela interface `Comestível` e as classes `Bolo` e `Torta`.

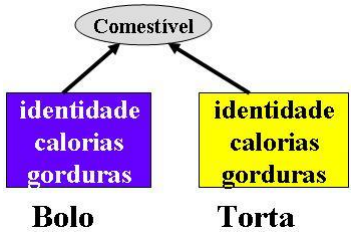


Figura 3.1 Hierarquia de `Comestível`

A obrigação de uma classe que implementa uma interface é dar corpo aos cabeçalhos dos métodos listados na interface. Membros de dados podem ser incluídos livremente na classe à medida que sejam necessários à sua implementação, como ilustram as declarações a seguir.

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

```
11     private double gorduras;
12     public Torta(double c, double g){ calorias = c; gorduras = g; }
13     public String identidade() { return "Torta"; }
14     public double calorias() { return calorias;}
15     public double gorduras() { return gorduras; }
16 }
```

Uma declaração da forma `class A implements I` estabelece um relacionamento **é-um** entre a classe `A` e a interface `I`. Este relacionamento autoriza o uso de objetos do tipo `A` onde, objetos do tipo `I` forem esperados, por exemplo, objetos do tipo `A` podem ser passados a funções que esperam parâmetros do tipo `I`.

O estabelecimento da relação **é-um** definida na hierarquia exibida na Figura 3.1 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  teste2 (String [ ] args){
6        Comestível b = new Bolo(150.0, 19.0);
7        Comestível t = new Torta(300.0, 56.0);
8        double z = g(b) + g(t);
9        System.out.println(b.identidade() + ":");
10       System.out.println("  " + b.calorias() + " kcal");
11       System.out.println("  " + b.gorduras() + "g de gorduras");
12       System.out.println(t.identidade() + ":");
13       System.out.println("  " + t.calorias() + " kcal");
14       System.out.println("  " + t.gorduras() + "g de gorduras");
15       System.out.println("Total de Calorias de = " + z);
16    }
17 }
```

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

Compartilhamento de Implementação

Uma mesma classe pode implementar várias interfaces. Essa classe 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 em mais de uma interface, somente um corpo de método é necessário para implementar esses cabeçalhos. Considere as seguintes interfaces:

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

```
6     double calorias( );
7     String identidade( );
8     double gorduras();
9  }
```

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:

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

Uma outra implementação dessas duas interface é a classe `Bolo` abaixo, que concretiza a ideia de que um `Bolo` é `Sobremesa` e também é `Comestível`:

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

As declarações de `Bolo` e `Fruta` acima criam a hierarquia de tipos da Figura 3.2:

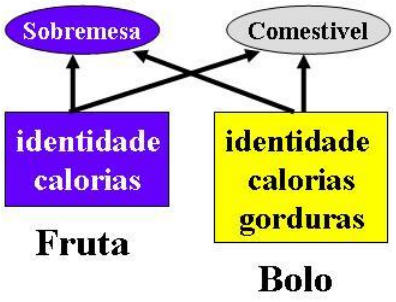


Figura 3.2 Hierarquia de `Sobremesa` e `Comestível`

Observe que as implementações das operações `identidade`, na linha `Bolo`.5, e `calorias`, na linha `Bolo`.6, definem os corpos das respectivas operações de ambas as interfaces, e que todo `Bolo` pode ser usado onde um `Comestível` ou uma `Sobremesa` forem esperados.

O programa `Usuário3` a seguir mostra o uso das implementações das interfaces:

```
1  class Usuário3 {
2     private String f(Sobremesa x) { x.calorias(); }
```

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

Uma classe, além de implementar zero ou mais interfaces, sempre estende **uma** outra classe. Se nenhuma extensão for especificada, a classe, por *default*, estende `Object`. Assim, os métodos definidos nas interfaces podem ser implementados diretamente na classe ou então herdados de outra classe.

O formato geral da declaração de uma classe é:

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

}
```

3.2 Hierarquia de Interfaces

Pode-se construir hierarquia de interfaces, fazendo uma interface estender outras. Nesse processo, todos os protótipos e constantes definidos nas interfaces base tornam-se elementos da interface derivada.

Protótipos de métodos idênticos são unificados na interface derivada. Constantes simbólicas herdadas tornam-se elementos da interface derivada. Os casos de conflitos de nomes de constantes são resolvidos via qualificação dessas constantes pelo nome da interface base correspondente.

A única situação de real conflito ocorre quando os protótipos de métodos herdados diferem apenas no tipo de retorno. Estas situações devem ser evitadas, pois o tipo de retorno não é usado, em Java, para distinguir chamadas de métodos, e, portanto, esses protótipos não podem ser implementados por um mesmo método.

O formato geral de definição de uma interface é:

```
[public] interface Nome [extends Interface1, Interface2, ... ] {
   protótipos de métodos públicos
   campos estáticos, públicos e finais
}
```

onde vê-se a possibilidade de se construir herança múltipla de interfaces.

As declarações abaixo exemplificam uma hieraquia simples de interfaces:

```
1 public interface Figura {
2    void desenha( );
3    void apaga( );
4    void desloca( );
5 }
```

```
6  interface Poligono extends Figura {
7     double perímetro( );
8  }
9  interface Retangulo extends Poligono {
10    double area( );
11 }
12 interface Hexagono extends Poligono {
13    double area( );
14 }
```

A hierarquia de tipos formada está na Figura 3.3.

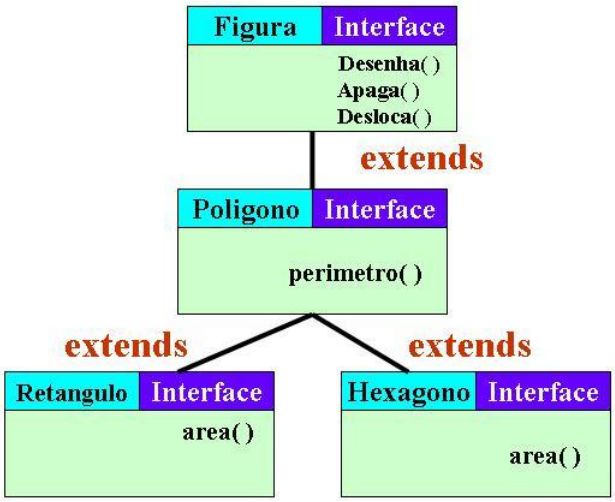


Figura 3.3 Hieraquia de Interfaces

As interfaces que formam uma hierarquia de tipos podem ser individualmente implementadas, por meio de classes concretas independentes, sendo que as classes que implementam elementos localizados 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].

As classes `Retângulo1` e `Retângulo2` mostram o esquema de duas implementações diretas do tipo `Retângulo`:

```
1  class MeuRetângulo implements Retangulo {
2     membros de dados
3     public void desenha( ) {...}
4     public void apaga( ) { ... }
5     public void desloca( )  { ... }
6     public double perimetro( ){ ... }
7     public double area( ) { ... }
8  }
9  class SeuRetangulo implements Retangulo {
```

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

```
10      membros de dados
11      public void desenha( ) { ... }
12      public void apaga( ) { ... }
13      public void desloca( ){ ... }
14      public double perímetro() { ... }
15      public double area() { ... }
16  }
```

A hierarquia de tipo completa é a da Figura 3.4, onde `MeuRetângulo` e `SeuRetângulo` são implementações das interfaces `Retângulo`, `Polígono` e `Figura`.

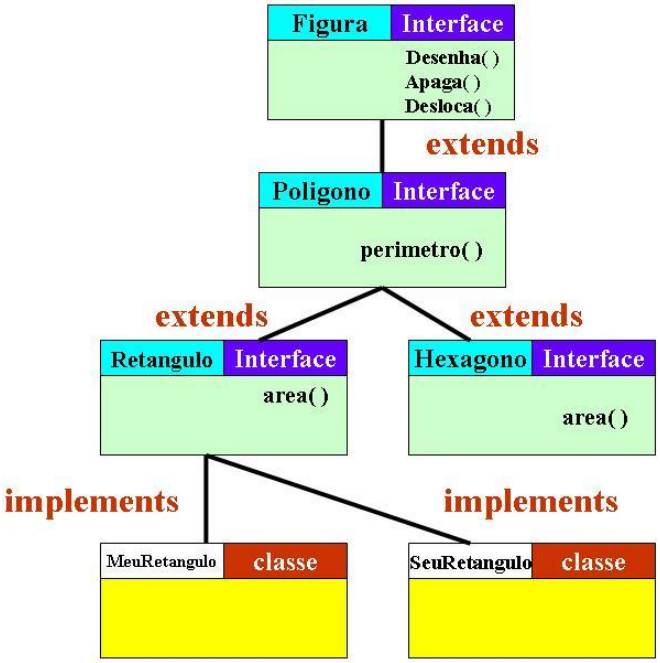


Figura 3.4 Hierarquia de Figuras

Na classe `Polivalente` apresentada a seguir, a referência `q` a um objeto do tipo `MeuRetângulo` é passada para os métodos `m1`, `m2` e `m3`, que esperam um parâmetro do tipo `Figura`, `Polígono` ou `Retângulo`, respectivamente, ilustrando o elevado grau de reúso que se pode alcançar com o uso de interfaces. Note que a chamada `m4(q)` será apontada como errada pelo compilador Java, porque `Hexágono` não é um superior hierárquico de `MeuRetângulo`.

```
1  public class Polivalente {
2     void m1(Figura f)   {...; f.desenha(); ... }
3     void m2(Polígono p) {...; p.desenha(); ...; p.perímetro();...}
4     void m3(Retângulo r){...; r.desenha(); ...; r.área(); ... }
5     void m4(Hexagono r) {...; r.desenha(); ...; r.área(); ... }
6     void m() {
7        Figura q = new MeuRetângulo();
8        m1(q);  // Correto
9        m2(q);  // Correto
```

```
10          m3(q);  // Correto
11          m4(q);  // Errado no tipo do parâmetro
12      }
13  }
```

3.3 Implementação Dual

A representação ideal de um tipo abstrato de dados pode depender 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 esse tipo de implementação dual, que é ilustrada a seguir com a implementação do tipo abstrato de dados `Complexo`, definido por:

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

Tradicionalmente, há duas representações para um número complexo: coordenadas cartesianas e coordenadas polares, conforme mostra a Figura 3.5.

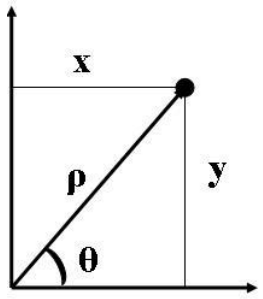


Figura 3.5 Representação do `Complexo x + yi`

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, pela fórmula de Euler, o número complexo $\rho \, (\cos\theta + i \sin\theta)$.

Considerando aplicações em 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 tal forma que cada operação possa escolher a representação que lhe for mais apropriada.

A classe `Dualidade` abaixo 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);
8        b.subtraia(c);
9        c.divida(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 }
```

O leiaute dos objetos alocados é mostrado na Figura 3.6.

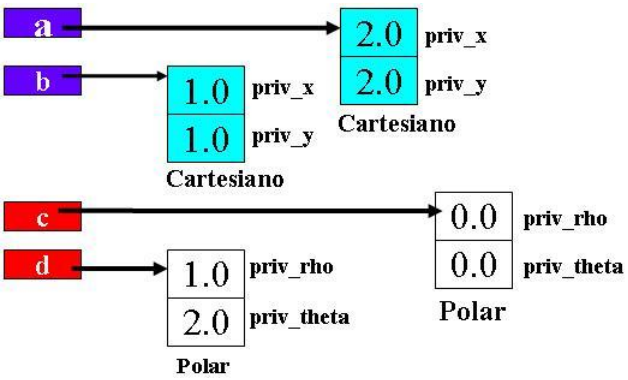


Figura 3.6 Leiaute dos Objetos de `Dualidade`

A classe concreta `Cartesiano` representa o número complexo pelos membros de dados privados `privX` e `privY`.

```
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;
13           c = new Cartesiano( );
14           c.privX = p.rho( )*cos(p.theta( ));
15           c.privY = p.rho( )*sin(p.theta( ));
16       } else c = (Cartesiano)w;
17       return c;
18    }
19    public String toString() {
20       StringBuffer b = new StringBuffer();
21       b.append(privX).append(" + ").append(privY).append('i');
22       return b.toString();
23    }
24    public void some(Complexo r) { ... }
25    public void subtraia(Complexo r) { ... }
26    public void muliplique(Complexo r) { ... }
27    public void divida(Complexo r) { ... }
28 }
```

As operações `Cartesiano.some` e `Cartesiano.subtraia` aceitam objetos do tipo `Complexo` como parâmetro, e, portanto, podem receber um objeto `Cartesiano` ou `Polar`. O método `Cartesiano.converta`, definido a partir da linha `Cartesiano`.9, é usado para assegurar-se que o número complexo dado pelo parâmetro está devidamente representado em coordenadas cartesianas, de forma que o cálculo do novo estado do objeto corrente possa ser feito em coordenadas cartesianas, como mostra o código:

```
1  public void some(Complexo r) {
2     Cartesiano rc = Cartesiano.converta(r);
3     privX = privX + rc.privX;
4     privY = privY + rc.privY;
5  }
6  public void subtraia(Complexo r) {
7     Cartesiano rc = Cartesiano.converta(r);
8     privX = privX - rc.privX;
9     privY = privY - rc.privY;
10 }
```

O mapa de alocação de objetos logo no início da operação da linha `Dualidade`.7 é mostrada na Figura 3.7.

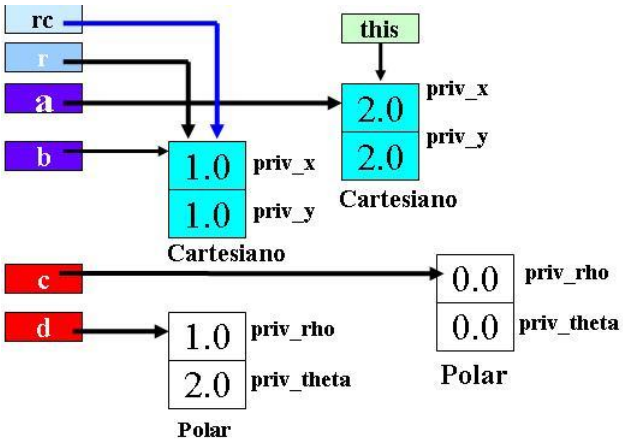


Figura 3.7 Alocação de Memória Para `a.some(b)`

As operações de multiplicação e divisão de números complexos são preferencialmente realizadas com os operandos representados em coordenadas polares, porque, como mostrado acima, essas operações são mais facilmente implementadas nessa representação.

```
1  public void multiplique(Complexo r) {
2     Polar p = Polar.converta(this);
3     p.multiplique(r);
4  }
5  public void divida(Complexo r)  {
6     Polar p = Polar.converta(this);
7     p.divida(r);
8  }
```

A seguir apresenta-se a implementação da classe `Polar`, que representa números complexos em coordenadas polares, por meio dos campos privados `privRho` e `privTheta`.

```
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    public static Polar converta(Complexo w) {
11       Polar p ; Cartesiano c;
12       if (w instanceof Cartesiano) {
13           c = (Cartesiano) w;
14           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     public void some(Complexo r)  { ... }
25     public void subtraia(Complexo r) { ... }
26     public void multiplique(Complexo r) { ... }
27     public void divida(Complexo r) { ... }
28  }
```

As operações `Polar.some` e `Polar.subtraia` são realizadas em coordenadas cartesianas, usando as correspondentes operações da classe `Cartesiano`.

```
1  public void some(Complexo r)  {
2     Cartesiano c = Cartesiano.converta(this);
3     c.some(r);
4  }
5  public void subtraia(Complexo r) {
6     Cartesiano c = Cartesiano.converta(this);
7     c.subtraia(r);
8  }
```

As operações de multiplicar e dividir de `Polar` são realizadas em coordenadas polares:

```
1  public void multiplique(Complexo r) {
2     Polar rp = Polar.converta(r);
3     privRho = privRho * rp.privRho;
4     privTheta = mod(privTheta + rp.privTheta,2*PI);
5
6  }
7  public void divida(Complexo r) {
8     Polar rp = Polar.converta(r);
9     privRho = privRho/rp.privRho;
10    privTheta = mod(privTheta-rp.privTheta,2*PI);
11 }
```

A alocação de objetos logo no início da operação da linha `Dualidade`.9 da classe `Dualidade` é mostrada na Figura 3.8.

Interfaces e Métodos Estáticos

Interfaces não aceitam declaração de protótipos de métodos estáticos, porque estes exigem sempre um corpo definido junto aos seus cabeçalhos. A compilação do programa abaixo aponta um erro de compilação na linha `I`.3 pela ausência do corpo do método estático `f`:

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

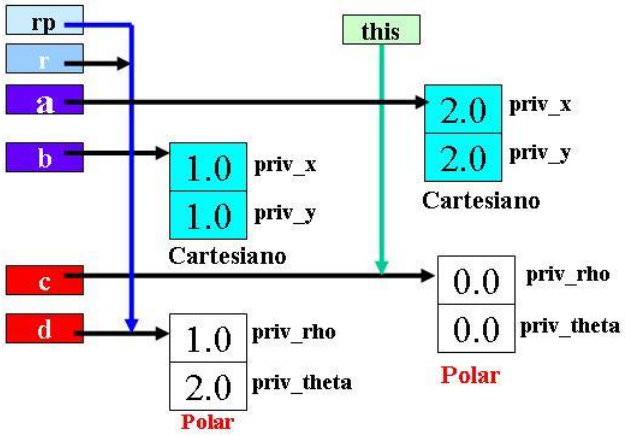


Figura 3.8 Alocação de Memória para `c.divida(d)`

Por outro lado, a declaração de `f` na linha B.3 da classe `B` não guarda relação alguma com o cabeçalho de método homônimo da interface `I`:

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

O programa abaixo apresenta um erro na linha C.3, porque interfaces não podem conter implementação de métodos:

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

3.4 Hierarquia de Classes

Ao declarar uma nova classe, pode-se aproveitar a declaração de membros de dados e de membros função de outra classe anteriormente definida. Este processo chama-se herança de classe e baseia-se na possibilidade de reúso da implementação de uma classe na definição de outra. Herança de classe é sobretudo uma técnica para construir novas classes, chamadas de classes derivadas ou subclasses, a partir de classes já existentes, que são ditas classes base ou superclasses.

A classe derivada herda todas as características de sua superclasse e pode adicionar outras. Herança de classe é um mecanismo para estender a funcionalidade de uma aplicação por meio da adição de funcionalidade à classe ascendente. Diz-se que a classe derivada **estende** a classe base.

A Figura 3.9 ilustra a construção das classes `Professor` e `Aluno` como subclasses ou extensão da classe `Pessoa`, que assim, descreve a parte comum de objetos `Aluno` e `Professor`.

Herança cria uma estrutura de subtipagem, onde objetos de um subtipo podem ser substitutos de objetos cujos tipos são seus supertipos. Isto ocorre porque os supertipos tem uma parte que é comum a todos os seus subtipos. Por exemplo, se objetos do

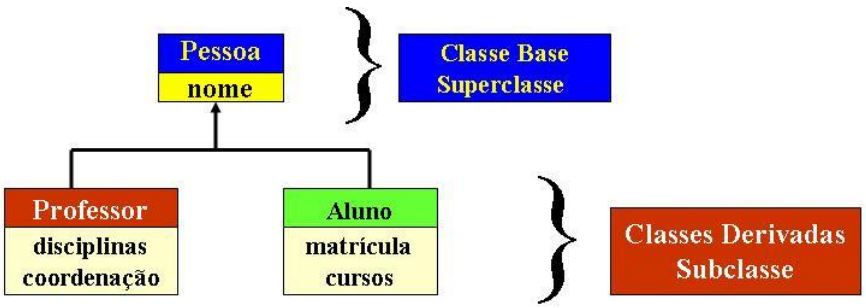


Figura 3.9

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, têm `nome`, `matrícula` e `cursos`. Note-se que a parte `nome` da representação dos objetos `Professor` e `Aluno` possui a mesma estrutura da parte correspondente de objetos `Pessoa`. Isto permite que objetos `Professor` e `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 `Professor` e `Aluno`. Portanto, `Professor` **é uma** `Pessoa`, bem como `Aluno` também **é uma** `Pessoa`. Genericamente, herança cria relacionamento **é-um** entre subclasses e sua respectiva superclasse.

A implementação da hierarquia da Figura 3.9 pode ter a seguinte forma:

```
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 relações de **especialização** e **generalização** entre as classes base e derivada, conforme ilustrada na Figura 3.10.

Não há limites no número de níveis hierárquicos. Figura 3.11 mostra uma nova hierarquia em que a classe `Empregado` foi introduzida para encapsular os elementos comuns de objetos `Professor` e `Funcionário`. Nesta estrutura tem-se que `Professor` e `Funcionário` são um `Empregado`, que, por sua vez, é uma **Pessoa**.

Reúso de Implementação

Para fins de analisar os recursos oferecidos pelo mecanismo de herança, considere a classe `Hora` a seguir, que define hora, com a precisão de segundos, e tem uma operação

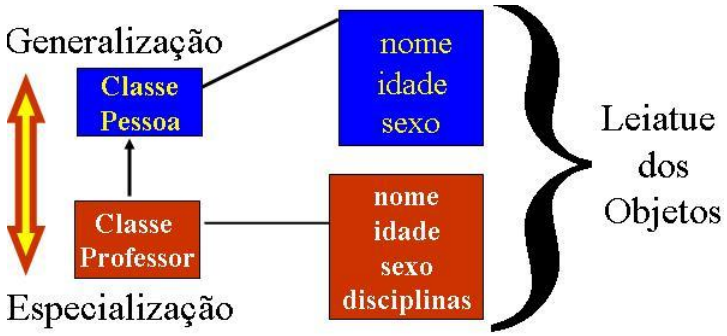


Figura 3.10 Especialização e Generalização

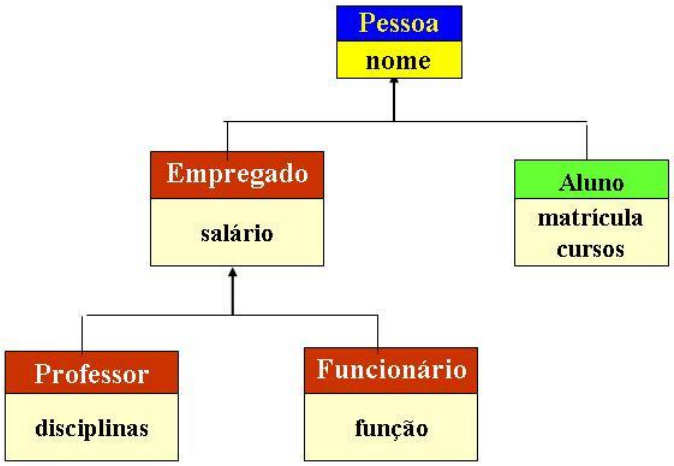


Figura 3.11 Hierarquia de Empregados

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  }
```

A classe `HoraLocal` 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(h + ":" + m + ":"  +  s " " +  local);
10    }
11 }
```

O programa `Horários` usa as classes `Hora` e `HoraLocal`, para criar objetos e exibe os horários anotados nestes objetos.

```
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( ); System.out.print(" "); t2.display( );
6     }
7  }
```

A saída do programa é: `11:30:50 10:26:10bsb`

A seguir, apresenta-se uma nova implementação de `HoralLocal`, com menos repetição de código, 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 `local`, sendo este declarado diretamente na classe. Como os campos herdados são privativos da classe `Hora`, 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 `HoraLocal`.4. Desta 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.

3.5 Tipo Estático e Tipo Dinâmico

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

O tipo estático é o tipo com o qual a referência foi declarada, 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 o tipo dinâmico pode mudar a cada atribuição ou passagem de parâmetro ocorrida no programa.

O tipo estático define as mensagens que podem ser enviadas a partir da referência, isto é, o tipo estático de uma referência informa os campos e métodos acessíveis a partir dela. O compilador Java somente permite acessos aos membros públicos declarados no tipo estático desta referência.

Por outro lado, o tipo dinâmico determina que método será executado para uma dada mensagem enviada via a referência, ou seja, a versão dos métodos 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 o tipo que define seu comportamento.

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.

O tipo estático e o tipo dinâmico da referência `this` são o tipo do objeto corrente. O tipo estático e o dinâmico da referência `super` são a superclasse da classe que a contém.

Para ilustrar o papel dos tipos de uma referência durante a execução de um programa, considere a classe `Ponto`:

```
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  }
```

e a classe `Pixel`, derivada de `Ponto`:

```
1  class Pixel extends Ponto {
2     private Color color;
3     public Pixel (double a, double b, Color c) {
4        super(a,b); color = c;
5     }
6     public Pixel ( ) { this(0,0, Color.white); }
7     public void clear( ) {this.color = null; super.clear( ); }
8     public void setcolor(Color c) { color = c; }
9  }
```

cujo leiaute de seus objetos é mostrado na Figura 3.12.

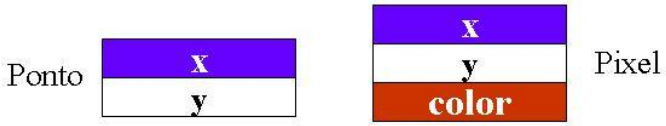


Figura 3.12 `Leiaute de Objetos Ponto e Pixel`

No programa abaixo `p1` e `p2` têm tipo estático `Ponto`, e `pixel`, `Pixel`.

```
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 `Janela`.3, `Janela`.4 e `Janela`.5, o tipo dinâmico de `p1`, `p2` e `pixel` são `Ponto`, `Pixel` e `Pixel`, respectivamente, como mostra a Figura 3.13. A execução dos comandos nas linhas `Janela`.6 e `Janela`.7 altera o estado dos objetos `p1` e `p2` para configuração da Figura 3.14. E após executar linha `Janela`.9, a configuração é a da Figura 3.15.

3.6 Execução de Construtoras

O operador `new`, de criação de objetos, aloca uma área de memória de tamanho suficiente para conter os campos do objeto e os inicia com valores *default*, conforme o tipo de cada um. A seguir, `new` inicia a execução da construtora da classe do objeto criado, que, em primeiro lugar, verifica se o primeiro comando da construtora não é uma chamada da construtora da superclasse, via o comando `super(...)`, e nem uma chamada a uma das construtoras irmãs, pelo comando `this(...)`. Se for este o caso, a construtora sem parâmetros da superclasse é chamada automaticamente, do contrário a execução obedece o que estiver escrito. Esse processo se repete para toda construtora chamada, até que a da classe `Object` tenha sido executada. Somente depois do retorno da chamada a 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 de execução de construtoras:

```
1  class A {
2     public A( ) { System.out.println("Passei na A()"); }
3  }
4  class B extends A {
5     public B (int a) {
6        super( );
7        System.out.println("Passei na B(a)");
8     }
9     public B( ) {
10       this(0,1);
11       System.out.println("Passei na B( )");
12    }
13    public B(int a, int b) {
14       System.out.println("Passei na B(a,b)");
```

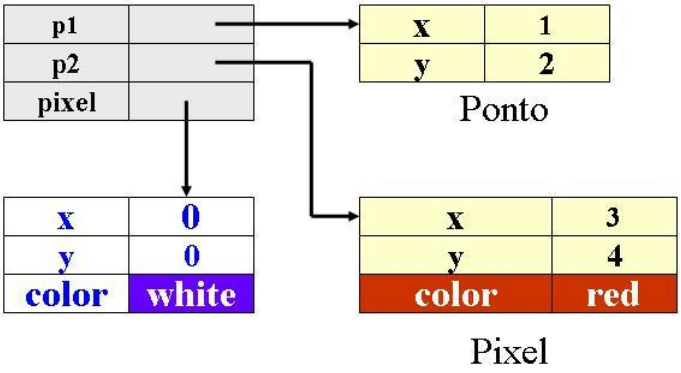


Figura 3.13 Configuração antes da linha `Janela`.6

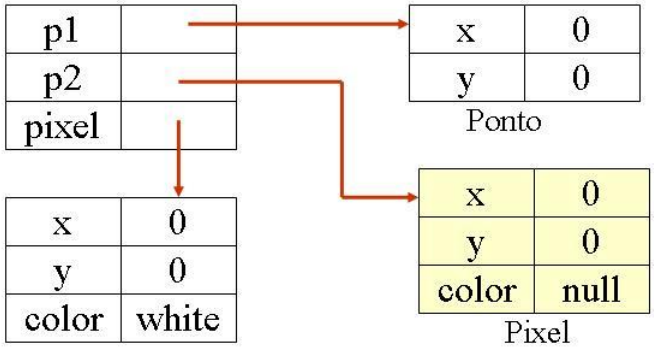


Figura 3.14 Configuração antes da linha `Janela`.8

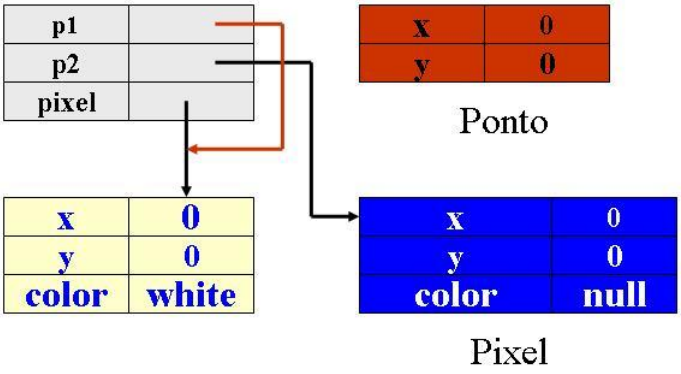


Figura 3.15 Configuração depois da linha `Janela`.9

```
15     }
16  }
17  public class C {
18     public static void main (String [ ]  args) {
19        System.out.println("Comecei na C");
20        B b1 = new B( );
21        System.out.println("Voltei na C");
22        B b2 = new B(1,2);
23        System.out.println("Terminei na C");
24     }
25  }
```

A execução da função `main` da classe `C` começa na linha 19. Na linha 20, após a alocação de um objeto `B`, a construtora da linha 9 é chamada, cujo primeiro comando, o `this(0,1)` da linha 10, transfere o fluxo de execução para a construtora da linha 13. Antes de executar o comando da linha 14, a construtora sem parâmetros da classe `A` é automaticamente chamada e, assim, o fluxo é transferido para a linha 2. Neste momento, a construtora da classe `Object`, superclasse de `A`, é executada, e, após o retorno desta construtora, imprime-se `Passei na A()`. Após a conclusão da execução da construtora de `A`, o controle volta à linha 14 para imprimir `Passei na B(a,b)`. A seguir retorna-se à linha 11, quando imprime-se `Passei na B()`, concluindo a alocação do primeiro objeto `B`.

Para a alocação do segundo objeto `B`, na linha 22, a construtora da linha 13 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. Neste momento, a construtora da classe `Object` é executada e, quando concluída, imprime-se `Passei na A()`. O controle então volta à linha 14 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
```

3.7 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 3.6. Somente quando o fluxo de controle retornar da execução destas 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`, que é declarada como sendo:

```
1 class Ponto {
2    int x = 10, y = 20;
3    Ponto( ) {
4       x = 1; y = 2 ;
5    }
6 }
```

Como o primeiro comando da construtora de `Ponto` não é uma chamada de construtora — `super` ou `this` —, o compilador Java transforma a classe `Ponto`, no momento da geração de código, para:

```
1 class Ponto {
2    int x = 10, y = 20;
3    Ponto( ) {
4       super( );       // inserido pelo compilador
5       x = 1; y = 2 ;
6    }
7 }
```

Similarmente, a classe `Pingo`, que foi declarada sem construtora,

```
1 class Pingo extends Ponto {
2    int cor= 0xFF00FF;
3 }
```

é transformada pelo compilador em:

```
1 class Pingo extends Ponto {
2     private int cor = 0xFF00FF;
3     public Pingo( ) {
4        super( );   // inserido pelo compilador
5     }
6 }
```

com a inserção da construtora *default*.

Com a finalidade de acompanhar os passos de iniciação, considere o seguinte programa:

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

O operador `new` da linha `Teste`.3 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 `Pingo`.4, a qual transfere o controle de execução para a linha `Ponto`.3.

O comando da linha `Ponto`.4 causa a chamada da construtora da classe `Object`. Quando a execução dessa construtora for concluída, os iniciadores de campo da classe `Ponto` são executados. Neste 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 `Pingo`.5, os iniciadores de campos da classe Pingo são executados, causando a iniciação do campo `cor` com o valor `0xFF00FF`. A seguir a execução da chamada $\widehat{\text{Pingo}}(\ )$ é concluída, completando 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 como a do programa:

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

O erro do programa, revelado pelo compilador com a mensagem: `B.java:8: cannot reference y before supertype constructor has been called`, é que o campo `y` da classe `B`, no momento em que é passado para a construtora, no comando `super(y)` da linha A.8, ainda não foi iniciado com o endereço do objeto `C`, criado no iniciador de campo da linha linha A.7. Isso ocorre porque o processamento de iniciadores de campo somente é feito após o retorno da chamada à construtora da superclasse.

Note que a tentativa abaixo de *enganar* o compilador também não funciona: o compilador Java sabiamente protesta com a mensagem:`Teste.java:3:cannot reference y before supertype constructor has been called`:

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

Por outro lado, o programa abaixo compila sem erro:

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

3.8 Atribuição de Referências

O polimorfismo das referências a objetos permite certa flexibilidade na atribuição de referências ou sua passagem como parâmetro, conforme define a regra:

> *Uma referência para uma classe base* `A` *pode ter o endereço de qualquer objeto de classe* `A` *ou qualquer 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)`, quando `p` tem a assinatura `T p(A a)`. Esta operação faz o tipo dinâmico de `a` ser `B`.

Entretanto, não se permite a atribuição `b = a`, ou passar a referência `a` para um parâmetro do tipo `B`, mesmo quando o tipo dinâmico de `a` for `B`.

Por razões de segurança, o compilador exige que sempre se escreva `b = (B)a`, ou `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`.

O polimorfismo das referências também se aplica a arranjos, mas cuidado adicional deve ser observado. Por exemplo se `v` tiver tipo `A[ ]`, onde `A` é uma classe, então `v` pode legitimamente apontar para instâncias de qualquer arranjo de elementos do tipo `B`, desde que B seja uma subclasse de `A`. Por exemplo, considere o programa `Atribuição` a seguir, onde o tipo estático de `v` é do tipo referência para arranjo de `A` e o de `r`, referência para arranjo de B, sendo este 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 aqui
13       r[0].x = 100;
14    }
15 }
```

A execução da função `main` da classe `Atribuição` até imediatamente antes da linha `Atribuição`.11 produz a alocação de objetos descrita na Figura 3.16.

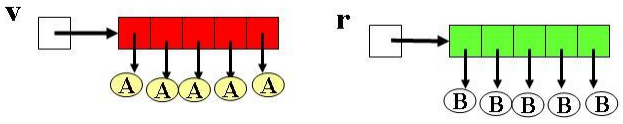


Figura 3.16 Vetores `v` e `r`

Os comandos nas linhas `Atribuição`.11 em `Atribuição`.12 alteram a alocação de objetos para a configuração da Figura 3.17.

A tentativa de execução da atribuição da linha `Atribuição`.12 causa o lançamento da exceção `ArrayStoreException` e consequente cancelamento do programa. O lançamento desta exceção visa impedir a situação mostrada na Figura 3.18, que seria gerada se a atribuição fosse permitida.

Figura 3.17 Vetores `v` e `r` Após `Atribuição`.11

Figura 3.18 Vetores `v` e `r` Após `Atribuição`.12

Note que no cenário da Figura 3.18 há um erro de endereçamento na tentativa de se ter acesso ao campo `x` pelo comando `r[0].x = 100` da linha `Atribuição`.13.

O exemplo a seguir indica o momento, linha `Teste`.7, em que este tipo de erro de armazenamento é detectado durante sua execução. Figura 3.19 mostra qual seria o problema de endereçamento, que ocorreria na linha `Teste`.10 e que é evitado com o lançamento da exceção.

```
1  class Ponto { int x, y; }
2  class PontoColorido extends Ponto { int cor; }
3  class Teste {
4     public static void main(String[ ] args) {
5        PontoColorido[ ] c = new PontoColorido[10];
6        Ponto[ ] p = c;
7        System.out.println(p[0] == null);
8        p[0] = new Ponto( );       // <== Erro de execução aqui
9        System.out.println("Nunca se chegará aqui");
10       c[0].cor = 255;   // <== Erro de endereçamento
11    }
12 }
```

A saída do programa é:

```
true
java.lang.ArrayStoreException
```

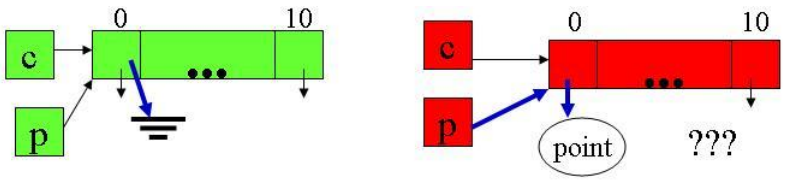


Figura 3.19 Configurações de `PontoColorido`

3.9 Especialização de Comportamento

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 classe base. 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-estática declaradas na classe base.

Funções estáticas e membros de dados não são redefiníveis, portanto suas redeclarações em subclasses são meras criações de homônimos 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 resolvido via sua qualificação pelo nome da classe que o contém.

Métodos declarados finais não podem ser redefinidos nas classes estendidas. As tentativas que vão de encontro a essa regra serão apontadas pelo compilador, como no programa:

```
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 de Compilação
8 }
```

Redefinição de Métodos

A visibilidade (`public`, `protected`, `private` ou *pacote*) de um método não pode ser reduzida na 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 métodos com tipos ou número de parâmetros diferentes dos do método herdado é permitido, mas, neste caso, não se trata efetivamente de redefinição de métodos da classe base, mas da introdução de um novo método homônimo, sobrecarregando um nome antigo.

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**.

No exemplo a seguir, 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 `A`. 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` a seguir mostra a flexibilidade e as restrições oferecidas pelo mecanismo de redefinição. Note que embora a redefinição de `g` retorne um objeto do tipo `B`, o valor retornado por `x.g()` é sempre suposto ser do tipo estático de `x`, pois não se pode garantir em tempo de execução o que será efetivamente retornado.

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

Redeclaração de Membros de Dados

Em Java, membros de dados não podem ser especializados. Somente membros função são especializáveis. A redeclaração de um campo da superclasse na subclasse que a estende é permitido, mas apenas tem o efeito de criar um novo campo na subclasse, a qual passa a ter dois campos com mesmo nome: o herdado e o outro localmente declarado. O nome local oculta o nome herdado em referências sem qualificação explícita. Em outros casos, os conflitos de nome são resolvidos pelo tipo estático da referência. O exemplo abaixo 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  }
```

Se o método `h` da classe `A` for executado, o campo `s`, ou implicitamente `this.s`, referenciado na linha `A`.3 é o declarado na linha `A`.2. Por outro lado, se for o `h` da linha `B`.7, o campo `s` é o da linha `B`.6.

No caso de referências explicitamente qualificadas, como as das linhas `OcultaDados`.6 e `OcultaDados`.7, 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();
4       A a = b;
5       a.h( ); b.h( );
6       System.out.print(" a.s = " + a.s);
7       System.out.print(" b.s = " + b.s);
8    }
9 }
```

O programa `OcultaDados` produz a saída: `B: B B: B a.s = A b.s = B`, e Figura 3.20 detalha o leiaute dos objetos do programa `OcultaDados`.

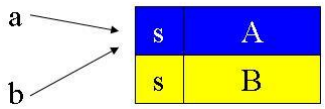


Figura 3.20 Leiaute dos Objetos de `OcultaDados`

3.10 Associação Dinâmica

Especialização de classes pode produzir mais de uma redefinição para uma mesma operação. Desta 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 estas informações para validar toda referência qualificada a membros do objeto. 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 este problema é chamado **ligação dinâmica** de operações.

Para ilustrar o funcionamento do mecanismo de ligação dinâmica, considere a hierarquia definida pelas classes `A`, `B` e `C` apresentadas a seguir.

```
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` sejam traduzidos para as funções `Fa`, `Ga` e `Pa`, codificadas em *bytecode* por funções cujas assinaturas, na linguagem **C**[22], correspondem às mostradas nos comentários das linhas de Â.3 a Â.5.

Para cada classe encontrada no programa, independentemente do número de suas ocorrências, carrega-se durante a execução o 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 **vmt** Tabela de Métodos Virtuais. Para a classe `A` definida acima, o objeto `Class` alocado tem o formato mostrado na Figura 3.21, onde a tabela **vmt** contém entradas para `f` e `g`.

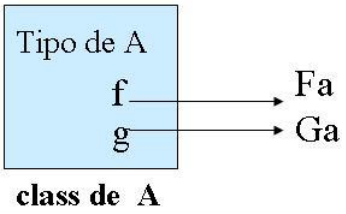


Figura 3.21 Descritor da Classe A

Na tabela **vmt** são alocados os endereços da primeira instrução de cada método redefinível da classe. Os métodos redefiníveis são numerados a partir de zero, e o número de cada um é usado para indexá-lo na tabela **vmt**. Esta numeração deve ser mantida consistente em toda a hierarquia da classe. No exemplo, `f` é associado a `0`, e `g`, a `1`. Métodos privados, estáticos ou finais nunca são incluídos na **vmt** da classe. Em geral, o código *bytecode* destes métodos são alocados junto ao descritor da classe, em local conhecido pelo compilador.

O programa `M1` abaixo aloca um objeto do tipo `A` e atribui seu endereço à `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 Figura 3.22.

```
1 public class M1 {
2    public static void main(String[ ] args) {
3       A r = new A( );
4       r.f(5);    // ( *(r->vmtr[0]) )(r,5);
5       r.g(5);    // ( *(r->vmtr[1]) )(r,5);
6    }
7 }
```

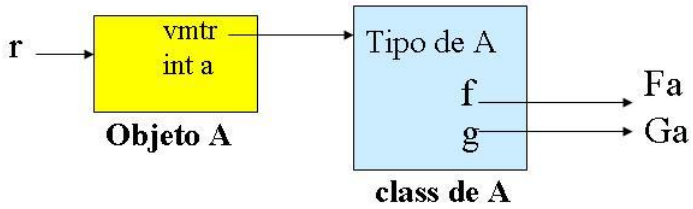


Figura 3.22 Leiaute dos Objetos de `M1`

Os comandos das linhas `M1.4` e `M1.5` são traduzidos para instruções em *bytecode* com semântica equivalente aos códigos na linguagem **C**[22] mostrados nos respectivos comentários, onde vê-se que a execução do comando `r.f(5)` segue os seguintes passos:

1. segue-se a referência `r` para ter acesso ao objeto alocado;

2. o objeto alocado fornece o endereço do descritor da sua classe;
3. o número de ordem de `f`, no caso `0`, é usado para indexar o vetor `vmtr` do descritor da classe e obter o endereço da primeira instrução da função `Fa` a ser executada;
4. efetua-se a chamada `vmtr[f](r,5)`, onde é passado como primeiro parâmetro o endereço do objeto corrente.

A tabela **vmt** de uma subclasse é construída a partir da tabela correspondente da superclasse, onde definem-se novos valores para as entradas correspondentes aos métodos redefinidos e, possivelmente, acrescentam-se novas entradas referentes a 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`. A redefinição de `g` em `B` é compilada para `Gb`, e no descritor da classe `B`, a entrada `g` refere-se a esta nova definição.

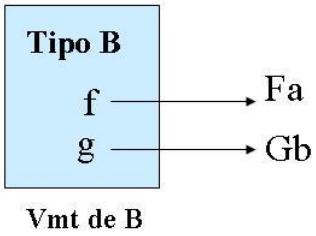


Figura 3.23 Descritor da Classe B

O leiaute de objetos do tipo `B` é exibido na Figura 3.23, 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 da correspondente na Figura 3.21.

O programa `M2` aloca um objeto do tipo `B` e cria para ele duas referências, `r` e `s`, do tipo estático `A` e `B`, respectivamente, conforme mostra a Figura 3.24. Ambas referências compartilham o mesmo objeto, portanto têm o mesmo tipo dinâmico. Consequentemente, os comandos `s.g(5)` e `r.g(5)` das linhas `M2.6` e `M2.7` causam a ativaçao da mesma versão compilada de `g`, a identificada por `Gb` na Figura 3.24.

```
1 public class M2 {
2    public static void main(String[ ] args) {
3       B s = new B( );
4       A r = s;
5       s.f(5);    // ( *(s->vmtr[0]) )(s,5);
6       s.g(5);    // ( *(s->vmtr[1]) )(s,5);
7       r.g(5);    // ( *(r->vmtr[1]) )(r,5);
8 }
```

Para completar o quadro, considere a classe `C` definida abaixo como uma extensão de `B`.

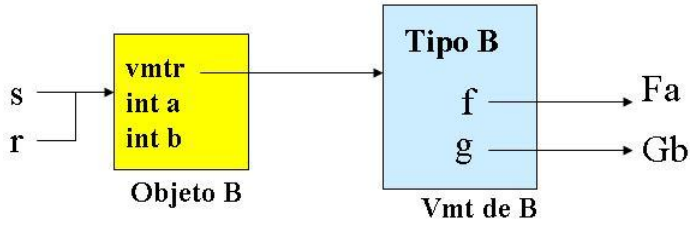


Figura 3.24 Leiaute dos Objetos de `M2`

```
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 Figura 3.25, herda a **vmt** do descritor de `B` e a ele acrescenta uma nova entrada, de índice `2`, para o novo método, `h`, declarado em `C`. Esta nova entrada é devidamente iniciada como o endereço de primeira instrução de `Hc`, que representa código *bytecode* de `h`.

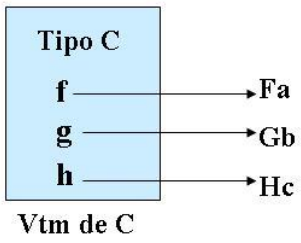


Figura 3.25 Descritor da Classe C

O programa `M3` 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.

```
1  public class M3 {
2     public static void main(String[ ] args) {
3        C t = new C( );
4        A r = t;
5        t.f(5);   // ( *(t->vmtr[0]) ) (t,5);
6        t.g(5);   // ( *(t->vmtr[1]) ) (t,5);
7        r.g(5);   // ( *(r->vmtr[1]) ) (r,5);
8        t.h(5);   // ( *(t->vmtr[2]) ) (t,5);
9     }
10 }
```

Qualquer chamada de método iniciada para as referências `t` ou `r`, com a configuração descrita na Figura 3.26, ativa a respectiva função compilada identificada pela **vmt** do descritor de `C`.

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. Este apontador identifica univocamente o tipo dinâmico de cada objeto. Os

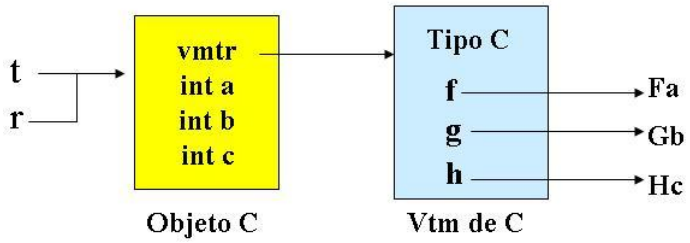


Figura 3.26 Leiaute dos Objetos de `M3`

métodos redefiníveis de cada classe tem 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`.

3.11 Redefinição de Membros

Para exemplificar as nuances do funcionamento do mecanismo de redefinição, ligação dinâmica e tipos dinâmicos de Java, considere o esquema de programa abaixo, onde pretende-se considerar os casos de o modificador (`<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 `A.3` é o seguinte:

- `private void increment( )`: Neste caso, o método da linha `B.11` não é uma redefinição do método homônimo da linha `A.3`. O fluxo de execução do programa

é o seguinte: H.14, H.15, H.16, A.4, A.5. A.3. Neste momento, o valor `A.i`, inicialmente `10`, é incrementado e o controle volta à linha A.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, estas duas classes pertencerem ao mesmo pacote, o método da linha B.11 é uma redefinição do método homônimo da linha A.3. Neste caso, o fluxo de execução do programa é o seguinte: H.14, H.15, H.16, A.4, A.5, A.11. Neste momento, valor de `B.i`, inicialmente `20`, é incrementado e o controle retorna à linha A.6, que imprime a resposta `i = 10`.
- `protected void increment( )` ou `public void increment( )`: O método da linha B.11 é uma redefinição do método homônimo da linha A.3, independentemente de qual pacote `A` e `B` estejam. O fluxo de execução do programa é o seguinte: H.14, H.15, H.16, A.4, A.5, A.11. Neste momento, valor de `B.i`, inicialmente `20`, é incrementado e o controle volta à linha A.6, que imprime a resposta `i = 10`.
- `static void increment( )`: Neste caso, o método da linha B.11 não é uma redefinição do método homônimo da linha A.3. O fluxo de execução do programa é o seguinte: H.14, H.15, H.16, A.4, A.5, A.3. Neste momento, o valor de `A.i`, inicialmente `10`, é incrementado, e a execução continua na linha A.6, que imprime a resposta `i = 11`.

3.12 Comparação de Objetos

Os estados de dois objetos cujos tipos pertencem a uma mesma hierarquia são iguais se e somente se ambos forem exatamente do mesmo tipo dinâmico, e portanto tiverem o mesmo leiaute e cada campo de um for exatamente igual ao que lhe corresponde no outro. Objetos de tipos distintos, mesmo pertencentes à mesma hierarquia, devem ser considerados distintos, independentemente de seu conteúdo.

O caráter polimórfico das referências e as restrições de visibilidade a membros de classe exigem um certo cuidado na implementação de um método, e.g., `igual`, que determina a igualdade do objeto corrente 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 igual(A a) {
5       if (a != null && this.getClass( ) == a.getClass( )) {
6             return (this.i == a.i);
7       } else  return false;
8    }
9 }
```

Durante a execução do método `igual` declarado na linha A.4, o objeto apontado pela referência `this` é garantidamente do tipo `A` ou de um de seus descendentes. Assim, o parâmetro `a` de `igual` tem que ter tipo dinâmico igual ao tipo de `this` para que possa haver igualdade entre os objetos apontados por `this` e `a`. O método `getClass` da classe `Object`, usado duas vezes na linha A.5, retorna o endereço do objeto descritor da classe

de seu objeto corrente. 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 objetos de mesmo tipo.

```
1  public class B extends A {
2     private int j;
3     public B(int i, int j) { super(i); this.j = j; }
4     public boolean igual(A a) {
5        if (a != null && this.getClass( ) == a.getClass( )) {
6            B b = (B) a;
7            return ( super.igual(a) && (this.j == b.j) );
8        } else return false;
9     }
10 }
```

A validade da conversão de tipo da linha B.6 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 dos campos `j` dos dois objetos. Para comparar os elementos da parte `A` de objetos do tipo `B`, deve-se ativar a versão do método `igual` definido na superclasse de `B`, na forma indicada na linha B.7.

3.13 Classe `Object`

Em Java, toda classe que não estende explicitamente outra, estende implicitamente a classe `Object`, cujos métodos são:

- `public boolean equals(Object obj)`
  Compara bit-a-bit objetos receptor e referenciado.

- `public int hashcode( )`
  Retorna o código hash do objeto receptor.

- `protected Object clone( )`
  Cria uma cópia ou clone do objeto receptor e retorna a referência ao objeto criado. Trata-se de uma cópia *rasa*, bit-a-bit.

- `public final Class getclass( )`
  Retorna o objeto do tipo `Class` que descreve a classe do objeto receptor. A classe `Class` é discutida em mais detalhes no Capítulo **??**.

- `protected void finalize( )`
  Finaliza o objeto receptor imediatamente antes de ele ser recolhido pelo coletor de lixo.

As operações acima são herdadas por todas as classes Java e portanto são aplicáveis a qualquer objeto criado em um programa Java.

3.14 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 pelo lado direito do comando de atribuição, no caso o endereço denotado pela variável `a` para a posição de memória denotada pelo lado esquerdo (`b`). Trata-se portanto de uma atribuição de uma referência e não do objeto referenciado:

```
1 class A {int x = 10; int y = 20 }
2 A a = new A( );
3 A b = a;
```

Assim, o efeito da execução da atribuição da linha 3 é ilustrado pela Figura 3.27.

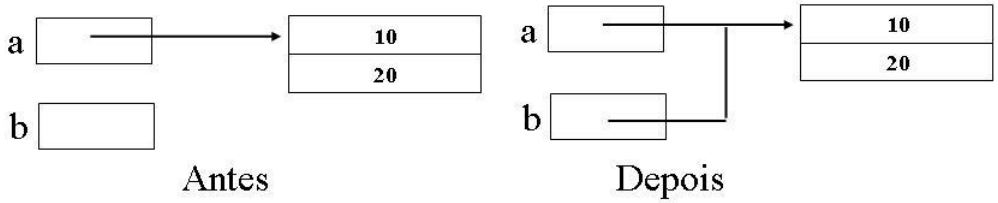


Figura 3.27

Há contudo situações que se deseja o efeito mostrado pela Figura 3.28.

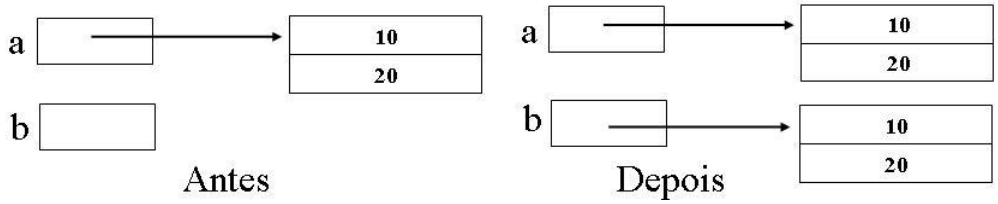


Figura 3.28

Note que neste caso uma cópia ou clone do objeto referenciado por `a` é efetivamente criado. Para obter este 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`, o qual é um método `protected` em `Object`. Este método `clone` 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 criada.

Como a cópia é *rasa* (**shallow copy**), isto é, ponteiros não são seguidos, a clonagem gera compartilhamento dos objetos apontados por campos do objeto copiado. Caso este não seja o efeito desejado, o programador deve redefinir o método herdado `clone` para propagar a operação de cópia aos objetos alcançáveis a partir do objeto copiado. Esta redefinição somente é permitida em classes que implementam a interface `Cloneable`. A violação desta regra causa o levantamento de uma exceção quando se requisitar clonagem.

Considere a classe `Dia` abaixo, definida como clonável:

```
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
10    public Object clone() {
11        return super.clone( );
12    }
13 }
```

A redefinição de `clone`, herdado de `Object`, apenas amplia a visibilidade do método para que ele possa ser usado por usuários de `Dia`. A criação da cópia é feita pela versão de `clone` herdada de `Object`.

O programa `TestaDia` a seguir 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 Figura 3.29, que os objetos apontados por `d1` e `d2` são compartilhados, mas `d3` referencia um objeto distinto, embora com os mesmos valores.

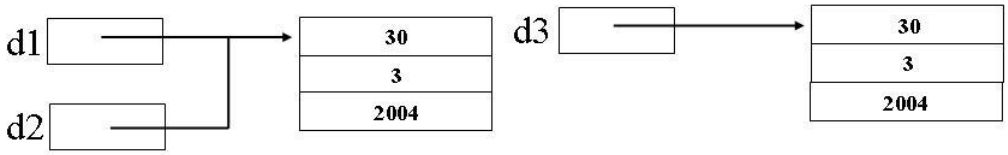


Figura 3.29 Clonagem de Dia (cópia rasa)

Considere agora a classe `Empregado` definida como clonável:

```
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
10     public Object clone() {
11        Empregado e = (Empregado) super.clone();
12        return e;
13     }
14  }
```

A execução do programa

```
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  }
```

gera o leiaute de objetos como o mostrado na Figura 3.30.

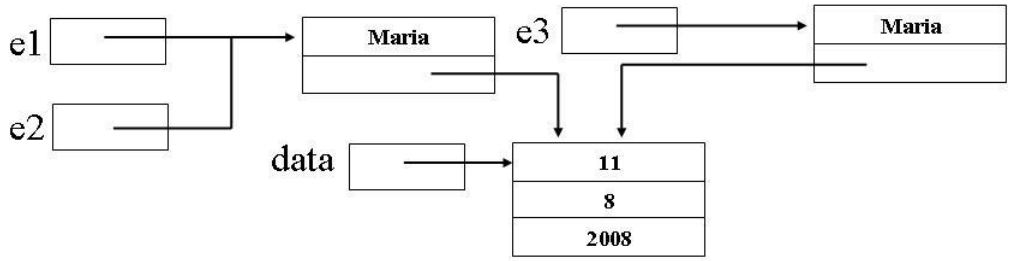


Figura 3.30 Clonagem de Empregado (cópia rasa)

Para evitar o compartilhamento da `dataContratação` pelos objetos clonados, deve-se redefinir o método `clone`, conforme mostrado a seguir:

```
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       e.dataContratacao = (Dia)dataContratacao.clone();
12       return e;
13    }
14 }
```

Para o mesmo programa `TestaEmpregado` acima, o leiaute dos objetos é o mostrado na Figura 3.31.

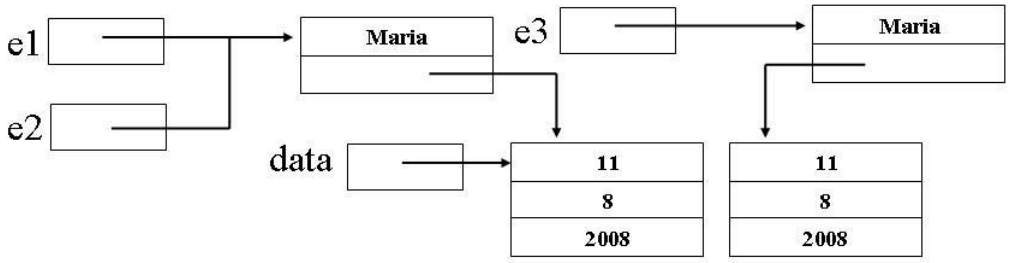


Figura 3.31 Clonagem de Empregado (cópia profunda)

3.15 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 a assinatura dos métodos e constantes por ela definidos. 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 seja compartilhada por todas as classes que a implementam, enquanto outros somente têm sua assinatura fixada. A solução para esta flexibilidade de expressão do nível de abstração desejado é o conceito de **classe abstrata**.

Classes abstratas, que são identificadas pela palavra-chave **abstract**, servem para definir um conjunto de membros comuns às suas subclasses. 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 classe.

Se uma classe definida como uma implementação de uma interface deixar de implementar qualquer um dos métodos cujas assinaturas estejam especificadas na interface, então a classe 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. Estas construtoras somente são usadas quando ativadas a partir de construtoras de subclasses. Normalmente a visibilidade de construtoras de classes abstratas deve ser `public`, `protected` ou `pacote`, para que seja possível definir-lhes subclasses.

Os 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 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 sobre a qual sabe-se apenas que deve ter um posicionamento, definido por coordenadas cartesianas, e uma cor. Deseja-se também que a área de formas específicas, a ser definidas por subclasses, possa ser calculada. Como não se sabe ainda que formas concretas podem ser definidas, o máximo que se sabe a respeito das operações para cálculo de área de formas ou para desenhá-las é sua assinatura, como mostra a seguinte definição da classe abstrata `Forma`:

```
1 import java.awt.*;
2 public abstract class Forma {
3    protected Color cor;
4    protected int x;
```

```
5     protected int y;
6     protected Forma(Color cor, int x, int y) {
7        this.cor = cor;
8        this.x = x;
9        this.y = y;
10    }
11    public abstract double área( );
12    public abstract void desenha( );
13 }
```

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 Circulo extends Forma {
2      private double raio;
3      public Circulo(Color c, int x, int y, double raio) {
4         super(c, x, y);
5         this.raio =  raio;
6      }
7      public Circulo( ) { this(Color.red, 0, 0, 1); }
8      public double área ( ) {
9         return 3.1416*raio*raio/2.0;
10     }
11     public void desenha( ) { ... }
12 }
13 public class Retangulo extends Forma {
14     private double largura;
15     private double altura;
16     public Retangulo(Color c, int x, int y, double a, double b) {
17        super(c, x, y);
18        this.largura = a;
19        this.altura  = b;
20     }
21     public void área( )  {
22        return largura*altura;
23     }
24    public void desenha( ) { ... }
25 }
```

Programa `Gráficos` a seguir usa a classe abstrata `Forma` para definir o tipo estático das referências `s1`, `s2` e `s3`, e as classes concretas `Círculo` e `Objetos` para criar os objetos desejados.

```
1  public class Gráficos  {
2     public static void main (String args) {
3        Forma s1 = new Retangulo(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("Area total = " + área);
9     }
10 }
```

Observe que, a partir das referências definidas, as operações particulares de cada um dos objetos são corretamente ativadas. Figura 3.32 mostra os objetos alocados.

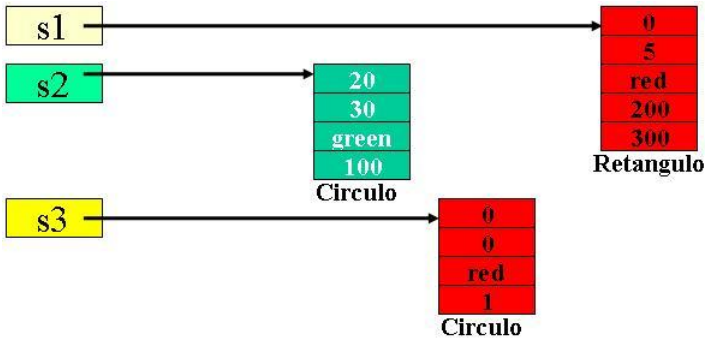


Figura 3.32 Objetos de Gráficos

3.16 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 foi 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 e campos

}
```

O mecanismo de estender classes e implementar interfaces podem gerar uma hierarquia de tipos, conforme mostra Figura 3.33. que 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 { ... }
```

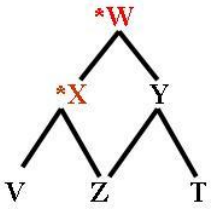


Figura 3.33 Hierarquia de Classes e Interfaces

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 isto, 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 a interface.

Protótipos de métodos homônimos, mas com diferentes assinaturas, especificados em várias interfaces, implementadas por uma mesma classe, são tratados como casos de sobrecarga.. Se, entretanto, eles tiverem a mesma assinatura, incluindo o mesmo tipo de valor de retorno, trata-se de dois protótipos da mesma função. Neste caso, um único método implementa todos protótipos correspondentes. Conflito somente ocorre quando mais de um protótipo têm a mesma assinatura, mas diferentes tipos de valor de retorno. Neste 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 apenas nos tipos das exceções (vide Capítulo 5) 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 abaixo, em que se têm objetos do tipo `Campo`, sobre os quais pode-se realizar as operações `edita`, `limpe` e `oculte`, objetos do tipo `Data`, sujeitos às operações `avance`, `defina` e `prazo` e objetos da classe `CampoDeData` que têm comportamento dual, porque são do tipo `Campo` e também do tipo `Data`. Sobre ele aplicam-se igualmente as operações de `Campo` e as de `Data`.

```
1 public class Campo {
2     ...
3     public Campo(...) { ...  }
4     public void edita(...) { ... }
5     public void limpe(...) { ... }
6     public void oculte(...) { ... }
7 }
8 public interface Data {
9      public void defina(... );
```

```
10      public void avance(...);
11      public int prazo(...);
12  {
13  public class CampoDeData extends Campo implements Data {
14       ....
15       public int prazo ( ... ) {...}
16       public void avance(... ) {...}
17       public void defina( ... ) {...}
18       public void mudeCor( ... ) { ... }
19  }
```

3.17 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 e também decorre da estrutura de tipos e subtipos criada pela hierarquia de classes e interfaces.

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.

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. Entretanto, o uso de interface permite reduzir parcialmente esta perda, herança simples atende à maior parte das aplicações e a proibição de herança múltipla na linguagem facilita a implementação do compilador.

Algumas linguagens, como C++[45] e Eiffel [33], permitem herança múltipla de classes, mostrando que sua implementação é factível, mas os mecanismos subjacentes necessários são complexos. Para compreender esta complexidade introduzida na implementação do compilador pela herança múltipla de classes, suponha, por um instante, que este recurso fosse permitido em Java, ou seja, suponha que a declaração

```
class C extends A, B { ... }
```

da Figura 3.34 fosse válida.

Suponha também que o código da linha 6 a 11 da Figura 3.34 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 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 `c` + Δ, onde Δ é o tamanho da parte `A` de objeto apontado por `c`. Portanto, imediatamente após a entrada no método `D.f`, tem-se `z = c` + Δ.

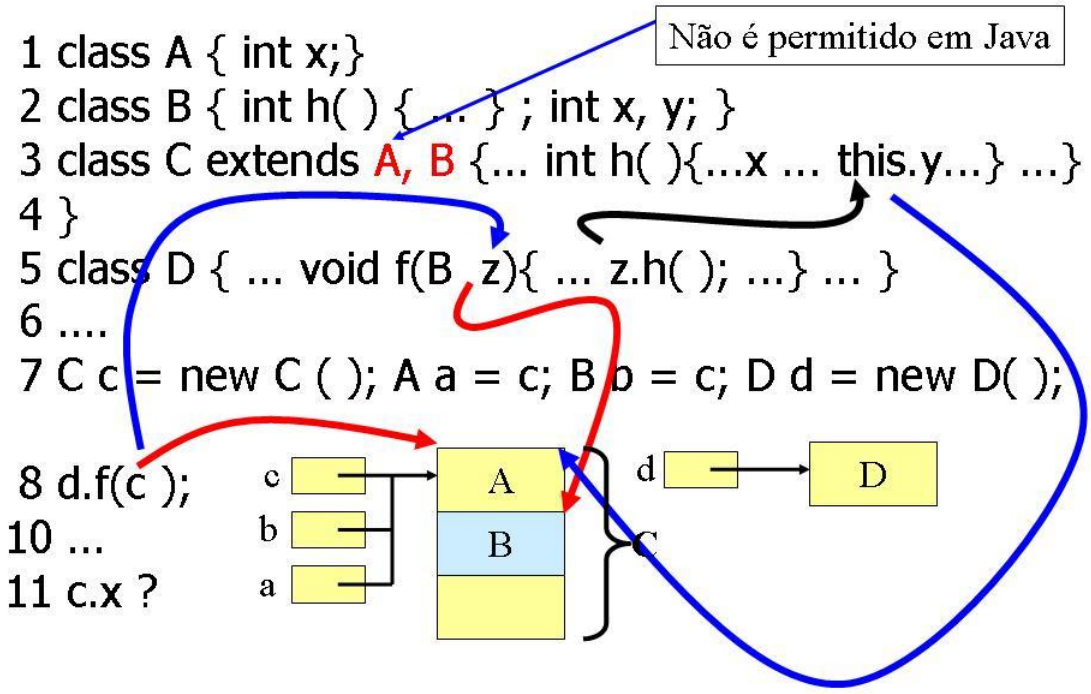


Figura 3.34 Objeto Corrente com Herança Múltipla

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`. Neste momento, o valor passado não deve ser o endereço da parte `B`, mas a do objeto integral apontado, i.e., o valor passado deve ser `z` - Δ.

3.18 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?

3.19 Notas Bibliográficas

O conceito de hierarquia de classes é bem discutido em praticamente todos os textos sobre Java, e em particular no livro dos criadores de Java, K. Arnold, J. Gosling e D.

Holmes, [2].

A técnica de implementação de tabelas de métodos virtuais e de herança múltipla é a utilizada no compilador de linguagem C++. Seus detalhes são descritos por M. Ellis e B. Stroustrup[13].

Capítulo 4

Polimorfismo

Polimorfismo é um conceito associado à idéia de entidades de um programa assumirem mais de uma forma ou tipo. Especificamente, um mesmo nome pode ser associado a diferentes entidades, e o compilador ou o sistema de execução sabem diferenciar as várias formas e implementações de cada nome. Nomes são usados via suas interfaces de forma abstrata e independente de implementação. Resumidamente, polimorfismo é a habilidade de se esconder diferentes implementações atrás de interfaces comuns. Polimorfismo manifesta-se em linguagens orientadas por objetos principalmente por meio de referências e de funções polimórficas.

4.1 Referências Polimórficas

Semântica de referência é um modelo de organização dos objetos de um programa baseado no foco na referência para um objeto, em vez de no seu valor, em operações, como atribuição e passagem de parâmetros, e na composição de objetos.

Referências de Java são os apontadores de linguagens imperativas tradicionais, como **C**[22], mas sem a sintaxe e a flexibilidade usuais de suas operações. Toda linguagem orientada por objetos depende dos recursos providos por apontadores para implementar de uma forma eficiente muitas construções, como composição de objetos e compartilhamento de dados. Sempre que mais de um objeto tiver 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 facilita a manutenção da coerência semântica dos dados, porque, quando a parte comum de vários objetos sofrer alguma modificação, não é necessário percorrer todos os objetos onde a informação está duplicada 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 usando semântica de valor.

Assim, objetos em Java são sempre manipulados via suas referências e, uma vez criados, nunca mudam de forma ou tipo. Todo objeto tem tamanho e leiaute pré-definidos no momento de sua alocação, que não mais são alterados durante a execução. Por isto, objetos são ditos monomórficos. Por outro lado, referências são **polimórficas**, porque, embora o tipo estático de uma referência seja fixo, seu tipo dinâmico, como o próprio nome indica, pode variar durante a execução do programa.

4.2 Polimorfismo de Inclusão

O mecanismo de herança, que permite a especialização de classes, cria uma relação **é-um** entre objetos, de forma que objetos da classe especializada podem ser usados onde os da superclasse forem esperados.

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 estes objetos define subconjuntos destes valores. A especialização é, portanto, um processo de descrever subconjuntos. Por exemplo, considere a classe `Médico` e sua subclasse `Cardiologista` apresentadas a seguir.

```
1  public class Médico {
2     "atributos de dados privados"
3     public Diagnóstico consulte(Paciente p) { ... }
4     public Remédio receite(Paciente p, Diagnóstico d) { ... }
5  }
6  public class Cardiologista 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, que são caracterizados pelas operações indicadas. A especialidade médica `Cardiologista`, 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.

Observe que referências declaradas com o tipo estático `Médico` podem apontar para objetos da subclasse `Cardiologista` e que subclasse (ou subtipo) corresponde a subconjunto enquanto classe (ou tipo) descreve o conjunto associado. 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 `Médico` e `Cardiologista`, referências são dotadas de **polimorfismo de inclusão**.

4.3 Paradoxo da Herança

No exemplo `Médico-Cardiologista`, 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 concedido pelo polimorfismo de inclusão das referências aos objetos associados a esta hierarquia.

Entretanto, herança também pode ser usada apenas com o objetivo de obter reúso de implementação. Nestes casos, conceitualmente subtipos não são criados, embora na prática subclasses ainda sejam vistas como subtipos, e o polimorfismo das referências a objetos das classes envolvidas ainda continua válido. Entretanto, paradoxalmente, nestes casos a correspondência conjunto-tipo e subconjunto-subtipo não é mais válida.

Para se compreender esse fenômeno, considere que um ponto no sistema cartesiano possa ser descrito pela 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( ); ajuste(x,y); exiba( );
9     }
10 }
```

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 esta sequência de ações seja indispensável para uma boa implementação de `mova`. 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 todo segmento é definido por dois pontos, pode-se definir a classe `Segmento` abaixo como uma especializacão de `Ponto`, por meio da adição de mais um par de coordenadas e redefinição apropriada de operações herdadas, conforme mostra-se em:

```
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` 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` devido à chamada da linha 5, o tipo dinâmico de `z` é `Ponto`, enquanto que a devida à da linha 6, 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); h(p);
6        r = new Segmento(5.0, 5.0, 15.0, 15.0); h(r);
7        ...
8     }
9     public static void main(...) { Figura f; ... f.g( ); ... }
10 }
```

O caráter polimórfico da referência `z` permite à operação `mova` utilizar a versão correta dos métodos `ajuste`, `apague` e `mostre`, 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, `Segmento` é visto como subtipo de `Ponto`, e, portanto, o primeiro pode ser usado onde o segundo 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 `Cardiologista`.

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, esta relação pode não ser válida. De um ponto de vista metodológico, o uso de herança simplesmente para atingir reúso deveria ser evitado, dando-se preferência a composição de objetos.

4.4 Funções Polimórficas

O conceito de polimorfismo das referências a objetos é estendido a funções, podendo neste caso ser classificado em **polissemia** ou **polivalência**.

Polissemia

Polissemia ocorre quando um mesmo nome de função é usado para designar funções distintas e possivelmente não-relacionadas. Estas funções possuem corpos independentemente definidos e se diferenciam no número ou tipo de seus parâmetros. 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. Este tipo de polimorfismo é comumente chamado de **sobrecarga**.

No exemplo abaixo, o nome `f` é polissêmico, porque tem seis significados diferentes, identificáveis a partir dos tipos estáticos de seus argumentos de cada invocação:

```
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 }
11 public UsoDeSobrecarga1 {
12    public static void main( String[ ] args) {
13       T t = new T( ); U u = new U( ); W c;  R m;
14       A a = new A( );  A b = new B( );
15       s1.f(t);         // função f da linha 2
16       c = a.f(t,u);    // função f da linha 3
```

```
17        m = a.f();       // função f da linha 4
18        b.f(t);          // função f da linha 7
19        c = b.f(t,u);    // função f da linha 8
20        m = b.f();       // função f da linha 9
21     }
22  }
```

Em Java, o tipo de retorno de métodos não é usado para diferenciar operações polissêmicas.

A redefinição de um método no processo de especialização de classes sobrecarrega o nome do método com uma nova definição, introduzindo novos significados a funções de mesma assinatura ao longo da hierarquia. Considere as declarações:

```
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 }
12 public class Sobrecarga2 {
13   public static void main( String[ ] args) {
14       T t = new T( ); U u = new U( );
15       A a = new A( ); A b = new B( ); A c = new C( );
16       a.f(t);        // função f da linha 2
17       a.g(u);        // função g da linha 2
18       b.f(t);        // função f da linha 6
19       b.g(u);        // função g da linha 2
20       c.f(t);        // função f da linha 9
21       c.g(u);        // função g da linha 10
22    }
23 }
```

Observe que a identificação da versão do método a ser invocado é feito em duas fases. Inicialmente, usam-se os tipos estáticos dos argumentos especificados, incluindo o do objeto receptor, para determinar o conjunto dos métodos candidatos. Depois usa-se o tipo dinâmico do objeto receptor para escolher a redefinição do método, dentre os candidatos, a ser ativada. Tipos estáticos são determináveis pelo compilador, mas a seleção do método pelo tipo dinâmico é feita durante a execução, conforme discutido na Seção 3.10.

Com polissemia, necessariamente há uma implementação distinta para cada novo significado que se esconde por trás de uma interface comum. O número de novos significados é finito e explicitamente definido caso a caso. Por esta razão, este tipo de polimorfismo é chamado de **polimorfismo ad-hoc** .

Polivalência

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 referência é polimórfica. Este é um polimorfismo distinto do discutido na seção anterior. Com polissemia mais de um significado é associado a um mesmo nome. Agora 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 essas funções têm definições únicas que são eficazes para parâmetros de diferentes tipos.

Ressalte-se que o polimorfismo de funções manifesta-se em duas dimensões: funções podem ser ao mesmo tempo polissêmicas e polivalentes.

Considere a classe `Polimórficos` abaixo, onde a hierarquia dos tipos usados na declaração dos parâmetros de suas funções está especificada na Figura 4.1:

```
1 public class Polimórficos {
2    void f1(Figura x) { ... x.desenha( ); ... }
3    void f2(Poligono y) { ... y.desenha( ); ... y.perímetro( ); ... }
4    void f3(Retangulo z) { ... z.desenha( ); ... z.area( ); ... }
5    void f4(MeuRetangulo w) { ... w.desenha( ); ... w.area( ); ... }
6 }
```

As funções `f1`, `f2` e `f3` e `f4` são todas polimórficas polivalentes, pois são aplicáveis a qualquer parâmetro cujo tipo dinâmico seja compatível com o especificado em cada caso.

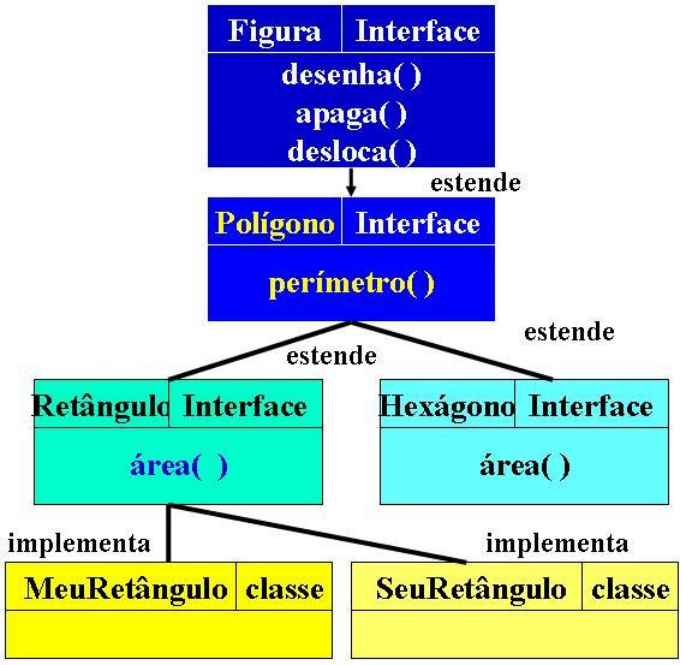


Figura 4.1 Hierarquia de Figuras

Identificação de Método Polissêmico

Na presença de hierarquia de classes, o mecanismo de resolução 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 passados, mas a relação hierárquica entre o tipo de cada parâmetro e tipo do argumento correspondente deve ser também usada. Para ilustrar estas novas regras, considere a seguinte hierarquia de classes:

```
1  public class A {
2     int x; ...
3  }
4  public class B extends A {
5     int y; ...
6  }
7  public class C extends B {
8     int z; ...
9  }
```

O programa `Teste1`, apresentado abaixo, imprime o texto `f(A)`, pois, embora o tipo dinâmico de `a` seja `B`, o seu tipo estático de `a` é `A`, o que causa a escolha da função `f` da linha 6, cujo parâmetro tem exatamente do mesmo tipo. Confira:

```
1  public class Teste1 {
2     public static void main(String[] args){
3        A a = new C();
4        f(a); //Imprime f(A)
5     }
6     public static void f(A a){
7        System.out.println("f(A)");
8     }
9     public static void f(B b){
10       System.out.println("f(B)");
11    }
12 }
```

No exemplo a seguir, a escolha recai sobre o `f` da linha 9, porque agora o tipo estático do argumento passado é `B`. Confira:

```
1  public class Teste2 {
2     public static void main(String[] args){
3        B b = new C();
4        f(b); //Imprime f(B)
5     }
6     public static void f(A a){
7        System.out.println("f(A)");
8     }
9     public static void f(B b){
10       System.out.println("f(B)");
11    }
12 }
```

No caso do programa `Teste3` a situação é diferente. O tipo estático do argumento da chamada `f(c)` da linha 4 não casa com o tipo do parâmetro formal da função definida na linha 6 e nem com o da função da linha 9. Neste 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 9 é a escolhida, pois na hierarquia de classes `C` $\rightarrow$ `B` $\rightarrow$ `A`, `C` está mais próximo de `B` do que de `A`.

```
1  public class Teste3 {
2     public static void main(String[] args){
3        C c = new C();
4        f(c); //Imprime f(B)
5     }
6     public static void f(A a){
7        System.out.println("f(A)");
8     }
9     public static void f(B b){
10       System.out.println("f(B)");
11    }
12 {
```

Quando se têm mais de um parâmetro, as regras exemplificadas acima devem ser aplicadas a todos os parâmetros. No exemplo a seguir, a chamada `f(a,b)`, da linha 5, aciona a função definida na linha 8, por ser esta 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 a execução da função definida na linha 11.

```
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 a seguir, o mecanismo de resolução de sobrecarga não funciona. Note que as chamadas das linhas 11 e 11 da classe `Teste5` são rejeitadas pelo compilador Java por serem ambíguas. Isto pode ocorrer quando se tem mais de um parâmetro. O mecanismo é aplicado a cada parâmetro, e o resultado deve permitir que a escolha da função seja feita. Entretanto, no caso em questão, para ambas chamadas, há duas possibilidades de escolha, que a regra de proximidade de tipo não resolve, pois cada parâmetro sugere uma solução distinta, gerando, assim, erro de compilação.

```
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);
7        f(b,a);
8        f(a,c);
9        f(b,c); //Erro de compilacao
10       f(c,c); //Erro de compilacao
11    }
12    public static void f(B b, A a){
13       System.out.println("f(B,A)");
14    }
15    public static void f(A a, B b){
16       System.out.println("f(A,B)");
17    }
18 }
```

4.5 Reúso de Funções

Polimorfismo está diretamente ligado a reúso de componentes de sofware. Funções polimórficas são mais reusáveis que funções monomórficas. Módulos que usam funções polimórficas são mais independentes de contexto, e portanto mais reusável.

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 feito 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 consequência direta 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 desta hierarquia.

Uso de 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 Figura 4.2, 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 os 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, esta perda de poder de expressão pode ser compensada pelo uso de interfaces.

Para ilustrar como esta perda de poder de expressão ocorre e como é solucionada, suponha que herança múltipla de classe seja permitida em Java e considere a modelagem apresentada nas declarações abaixo e resumida na hierarquia mostrada na Figura 4.2, na qual os diagramas apresentam no seu lado esquerdo os atributos privados da classe representada, e do lado direito as suas operações públicas.

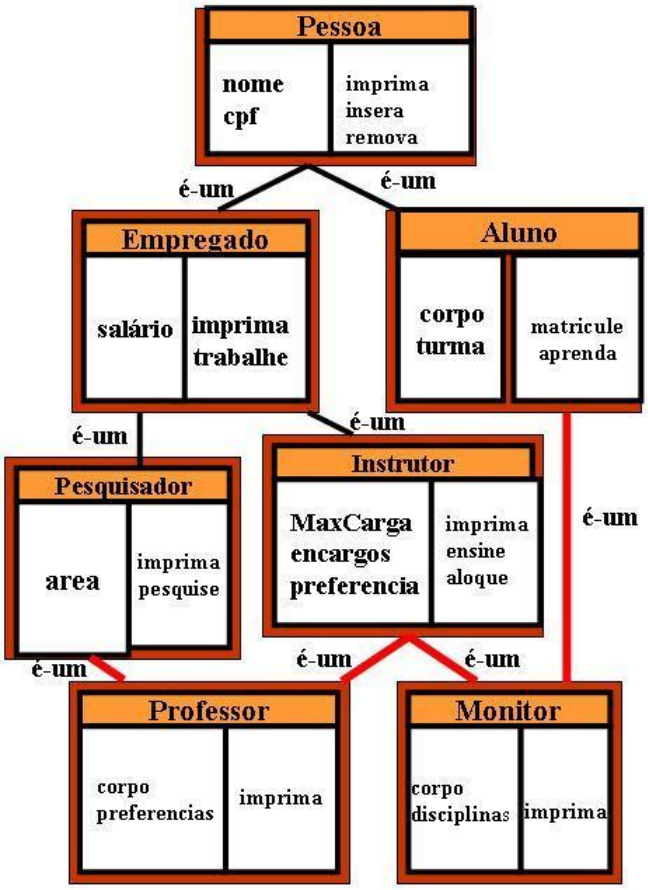


Figura 4.2 Sistema Acadêmico com Herança Múltipla

No sistema acadêmico de uma universidade, as pessoas são modeladas pela classe `Pessoa`, que encapsula seus elementos em comum:

```
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 }
```

Alunos e empregados são definidos como tipos especiais de classe `Pessoa`:

```
class Aluno extends Pessoa {
   private static LinkedList corpo;
   private int número;
   private Disciplina[ ] disciplinas;
   public Aluno(String nome, int número) {super(nome); ... }
   public void matricule(Disciplina disciplina) { ... }
   public void aprenda(...) { ... }
}
class Empregado extends Pessoa {
   private float salario;
   private String cpf;
   public Empregado(String nome, String cpf, float salário) {
      super(nome); this.cpf = cpf; ...
   }
   public void imprima( ) { ... }
   public void trabalhe( ) { ... }
}
```

Há dois tipos de empregados: os pesquisadores e os instrutores, sendo estes encarregados das aulas da Instituição:

```
class Pesquisador extends Empregado {
   private String área;
   public Pesquisador(String nome, String área,
                                   String cpf,float salário){
      super(nome, cpf, salário); ...
   }
   public void pesquise(Tema tema) { ... }
}
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) { ... }
}
```

Nas universidades, professores efetivos devem, além de lecionar, fazer pesquisas, sendo, portanto, tipos especiais das classes `Pesquisador` e `Instrutor`. Supondo que Java permitisse herança múltipla, define-se `Professor` com subtipo de `Pesquisador` e de `Instrutor` da seguinte forma:

```
class Professor extends Pesquisador, Instrutor {
   private static LinkeList corpo;
```

```
3     public Professor(String nome, String área,
4                                    String cpf, float salário){
5        super(...); ...
6     }
7     public void imprima( ) { ... }
8  }
```

Acima os detalhes das chamadas das construtoras das superclasses foram omitidas para não obscurecer o exemplo.

Como monitores são alunos que exercem funções de instrutores, herança múltipla surge novamente como uma solução natural. Assim, define-se `Monitor` como sendo 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,String número ) {
5        super(...); ...
6     }
7     public void imprima( ) { ... }
8  }
```

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 seguinte aplicação:

```
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 }
```

Note que os métodos de nomes `consulte`, `contrate` e `avalie` são polimórficos polivalentes: uma única implementação de cada um deles é usada para 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 para `Professor`.

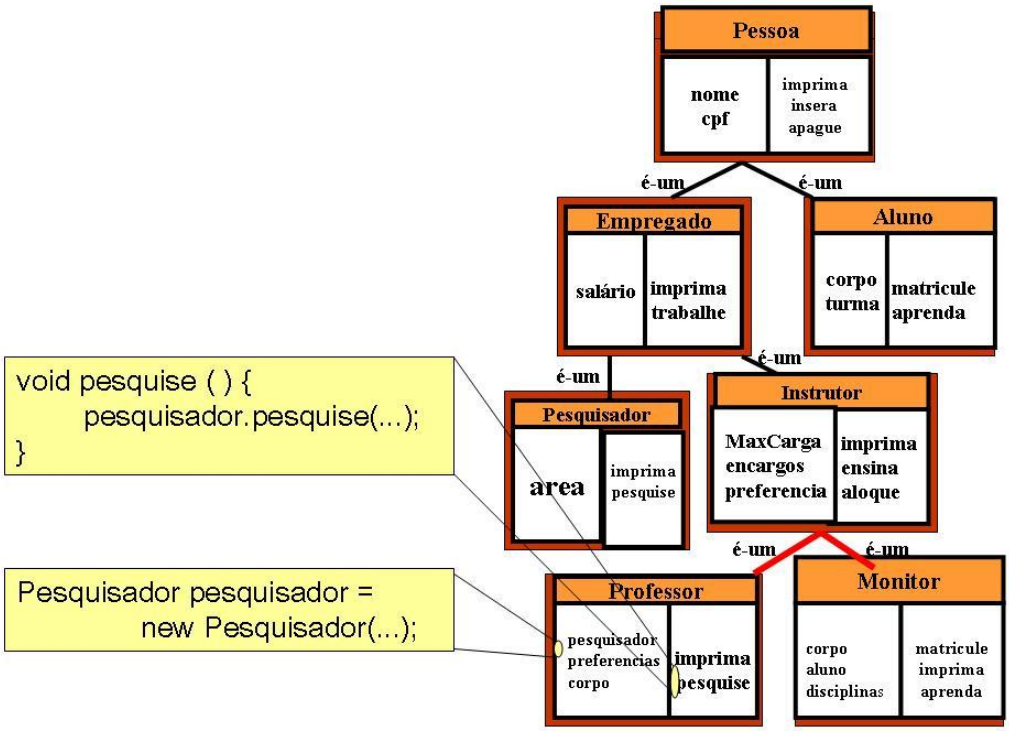


Figura 4.3 Sistema Acadêmico sem Herança Múltipla

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 mostra a Figura 4.3.

Com a eliminação da relação **é-um** entre `Professor` e `Pesquisador` e entre `Monitor` e `Aluno`, os métodos `consulte` e `avalie` perdem polivalência e devem ser duplicados para conservar a funcionalidade. Felizmente, a possibilidade de sobrecarregar nomes permite que o cliente, no caso o método `main`, fique inalterado. Confira:

```
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 reúso de implementação: campos e métodos de `Pesquisador`, por exemplo, não são mais herdados por `Professor`, que deve defini-los diretamente, por composição explícita. A replicação dos campos antes herdados de `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(...); }
```

Perda de reúso similar ocorre na eliminação da relação **é-um** entre `Monitor` e `Aluno`, e solução análoga para recuperar reúso deve ser usada.

Herança Múltipla com Interfaces

Interfaces podem ser usadas para restabelecer a perda de polivalência descrita no exemplo acima e evitar duplicação de código. Para isto, é necessário colocar objetos do tipo `Professor` e `Pesquisador` em uma mesma hierarquia aceitável pelo método `consulte`. E analogamente o mesmo deve ser feito com `Monitor` e `Aluno`. Para isto, deve-se inicialmente criar as interfaces:

```
public interface Pesquisador {
   void pesquise(Tema tema);
   void imprima( );
}
public interface AlunoI {
   void matricule( );
   void aprenda( );
}
```

A seguir altera-se a hierarquia de tipos, modificando os cabeçalhos das declarações de `Pesquisador`, `Professor`, `Aluno` e `Monitor` para:

```
public class Pesquisador extends Empregado implements PesquisadorI{...}
public class Professor extends Empregado implements PesquisadorI{...}
public class Aluno extends Pessoa implements AlunoI {  ... }
public 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 Figura 4.4.

Por fim, mudam-se os tipos dos parâmetros dos métodos `consulte` e `avalie` para `PesquisadorI` e `AlunoI`, respectivamente, passando-se a ter os métodos

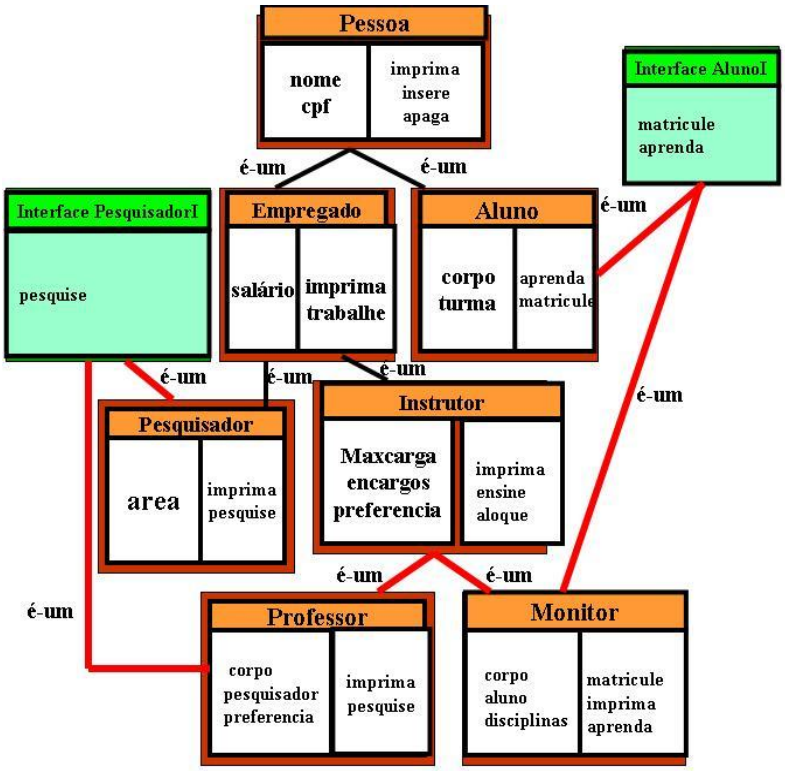


Figura 4.4 Sistema Acadêmico com Interfaces

```
void consulte(PesquisadorI x) { ... }
void avalie(AlunoI z) { ... }
```

no lugar dos quatro métodos definidos da linha 2 à linhas 5 da classe `Aplicação2`. Note que os corpos desses métodos não precisam ser modificados. Apenas o tipo do parâmetro é que deve ser alterado.

Claramente o recurso de interface permite uma modelagem quase tão natural quanto ao oferecido pelo mecanismo de herança múltipla, porém a um custo muito menor.

4.6 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.

4.7 Notas Bibliográficas

O termo **polimorfismo**, no contexto de linguagens de programação, foi introduzido por Cristhropher Strachey em 1967 [43, 44].

Strachey classificou as manifestações de 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[3] modificaram a classificação de polimorfismo de Strachey, criando a categoria de polimorfismo **universal** para generalizar o **paramétrico**. Nesta nova classificação, o polimorfismo que se manisfesta em linguagens de programação pode ser:

- *ad-hoc*:
  - sobrecarga
  - coerção
- universal:
  - paramétrico
  - inclusão

Capítulo 5

Tratamento de Falhas

Métodos de bibliotecas ou métodos-servidor têm meios para detectar, durante sua execução, situações de erro que lhes impedem de cumprir sua tarefa, mas geralmente não sabem o que fazer nestas situações. Por outro lado, métodos-cliente 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 este problema consiste em separar a detecção do erro de 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 deste código após cada chamada ao método para tomar as providências necessárias. Entretanto, esta 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. Isto pode ser evitado programando-se tratamento de erros por meio de lançamento e captura de exceções, que resolvem este problema e ainda ajuda aumentar o grau de reusabilidade de módulos.

Com o uso de exceções, métodos-servidor 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-cliente perceberem, sem necessidade de testes, se houve ou não falha na execução do método-servidor e processar cada tipo de retorno de forma apropriada.

5.1 Declaração de Exceções

Uma exceção em Java é representada por um objeto de um tipo da hierarquia da classe `Throwable`, mostrada na Figura 5.1. Para se criar uma classe de exceção, deve-se estender uma das classes da família de `Throwable`. As exceções do tipo `Error` e `RunTimeException` são chamadas exceções **não-verificadas** (**unchecked**). As extensões de `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., levantar exceções para ser tratadas por métodos que o invocam, tem o formato:

```
<modificador> T f(lista de parametros) throws <lista de exceções>
```

A lista de `throws` é opcional, mas um método sem essa lista 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 excecão `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 Figura 5.1, o usual nas aplicações é criá-las como exceção verificada, estendendo a classe `Exception`.

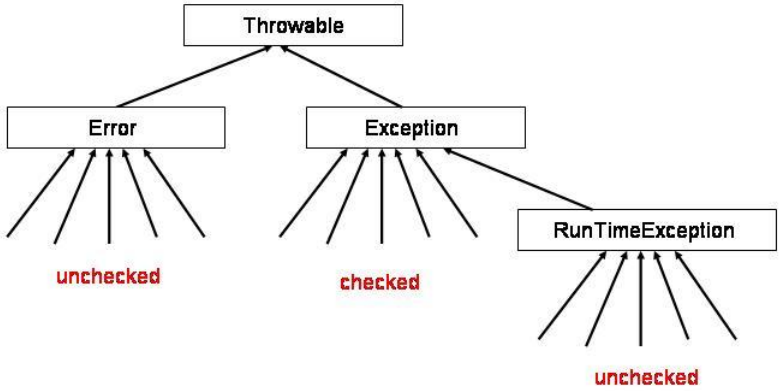


Figura 5.1 Hierarquia das Exceções

5.2 Lançamento de Exceções

Um método indica que falhou, i.e., que encontrou uma situação de erro intransponível, que o impede de avançar na computação, lançando uma exceção, via o comando `throw new X(...)`, o qual cria um objeto 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 as excecões. Todo `catch` captura, na ordem de sua ocorrência no comando `try`, os objetos de exceção cujo tipo tem a relação **é-um** com o seu parâmetro.

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:

```
1 try { comandos  }                 // bloco b1
2 catch(Excecao1 e) { comandos }    // bloco b2
3 catch(Excecao2 e) { comandos }    // bloco b3
4 ...
5 catch(excecaoN e) { comandos }    // bloco b4
6 finally { comandos }              // bloco b5
```

onde as cláusulas `catch` e `finally` podem ser omitidas.

Um comando `try` funciona como um comando bloco. Quando ativado na sequência normal de execução, seu corpo, formado pelo bloco `b1`, da linha 1, é executado na sequência do fluxo de controle e depois executa-se o bloco `b5`, 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 5.3.

Se durante a execução do bloco `b1` 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 à 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 desta cláusula, o bloco `b5`, 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 sucessivas vezes até que, na sequência inversa das chamadas, se encontre um `try` ativo. Neste momento, retoma-se à busca da cláusula `catch` desejada nesse `try` ativo, 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 destes comandos. A classe `E` é uma classe de exceção e o método `g` de `A` pode ou não lançar a exceção `E`.

```
1 class E extends Exception {
2    public int v;
3    public E1(int v) { this.v = v; }
4 }
5 class A {
6    void g( ) throws E {
7       int k;
8       ... throw new E(k); ...
9    }
10   ...
11 }
```

O programa `B` inicia-se com a execução dos comandos da linha `B`.4, inicia-se a execução do bloco da linha `B`.5. A chamada `a.g()` leva à execução da linha `A`.8, 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 B.5, e o restante do corpo do `try` é concluído. A seguir, o bloco `finally`, da linha B.7, é executado concluindo-se o comando `try`, e o fluxo segue normalmente na linha B.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 }
```

Se durante a execução da linha A.try2.8, comando `throw` for executado, a exceção `E` é lançada, e passada ao parâmetro do `catch` da linha B.6, e a execução continua no bloco de comandos desta linha. A seguir, o bloco do `finally`, da linha B.7, é executado para concluir o comando `try`, e o fluxo segue normalmente na linha B.8.

Considere a classe de exceção `Indice`, definida por:

```
1  class Indice extends Exception {
2     public int i;
3     Indice(int i) { this.i = i; }
4  }
```

A implementação de `MeuVetor` prevê erros de indexação, que devem ser tratados adequadamente pelo cliente dos métodos `MeuVetor.elemento` e `MeuVetor.atribui`:

```
1  class MeuVetor {
2     private int [ ] p; private int tamanho;
3     public void MeuVetor(int n) {
4        p = new int[n]; tamanho = n;
5     }
6     public int elemento(int i) throws Indice {
7        if ( 0 <= i && i < tamanho )
8            return p[i] ;
9        throw new Indice(i);
10    }
11    public void atribui(int i, int v) {
12       if ( 0 <= i && i < tamanho )
13           p[i] = v ;
14       throw new Indice(i);
15    }
16 }
```

A execução linha a linha do programa abaixo segue a ordem: linhas C.20, C.21, C.3, Indice.4, C.4, C.5, C.6 (imprime `Ponto A`), C.7, C.17, MeuVetor.7, MeuVetor.9 (a exceção `Indice` é lançada), C.11 (imprime `Indice Errado = 10`), C.12, C.14 (imprime `Valor de k = 0`), C.22 (imprime `Chegou aqui`) e C.23.

```
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    int g(MeuVetor v) throws Indice  {
17        return v.elemento(10);
18    }
19    public static void main(String[] args) {
20       C c = new C( );
21       c.f();
22       System.out.println("Chegou aqui");
23    }
24 }
```

O exemplo a seguir mostra uma implementação da classe `SeuVetor` com a previsão de duas condições de erros, que podem ocorrer no momento da criação de objetos da classe, quando o tamanho especificado para o vetor for negativo, e na tentativa de recuperação de elementos do vetor armazenado no objeto fora de seus limites. As exceções a ser levantadas nestes casos são `Tamanho` e `Indice`, respectivamente.

```
1  class Indice extends Exception {
2     public int i;
3     Indice(int i) { this.i = i; }
4  }
5  class Tamanho extends Exception {
6     public int t;
7     public Tamanho(int t) { this.t = t; }
8  }
9  class SeuVetor {
10    private int [ ] p; private int tamanho;
11    public SeuVetor(int n) throws Tamanho {
12       if ( n < 0 )
13           throw new Tamanho(n);
14       p = new int[n]; tamanho = n;
15    }
16    public int elemento(int i) throws Indice {
17       if ( 0 <= i && i < tamanho )
```

```
18              return p[i] ;
19         throw new Indice(i);
20      }
21      public void atribui(int i, int v) {
22         if ( 0 <= i && i < tamanho )
23             p[i] = v ;
24         throw new Indice(i);
25      }
26  }
```

O programa usuário da classe `SeuVetor` prevê o tratamento apenas da exceção `Tamanho`. O tratamento da exceção `Indice` é delegado à **JVM**.

```
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){System.out.print("Tamanho = " + e.t);}
16        System.out.print(" e tudo sob controle");
17     }
18 }
```

O levantamento de `Indice` na linha 19, quando `n = 10`, por exemplo, causa o cancelamento do programa. No caso de `n<0`, o programa encerra sua execução normalmente depois de imprimir `Tamanho = -10 e tudo sob controle`.

Todo método que pode falhar na execução de seu objetivo deve sinalizar sua falhar via exceção. Ressalte-se que iniciadores de campos, contidos nas declarações, não devem fazer chamadas de 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.

5.3 Cláusula `finally`

Na programação é bastante comum usar comandos para 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. Isto 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 este problema é a cláusula `finally` que abriga o bloco de comandos que sempre será executado após o término normal ou via exceção do bloco do `try`. Esta garantia faz da cláusula `finally` o local ideal dos comandos do tipo *desfazer*.

Para ilustrar o funcionamento de `finally`, considere o programa abaixo:

```
1  public class Finally {
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;
10       System.out.print("x = " + x);
11       try{ x = 11;
12             System.out.print("x = " + x);
13             throw new Exception();
14       }
15       catch(Exception e){
16            x = 12;
17            System.out.print("x = " + x);
18       }
19      finally {
20            x = 13;
21            System.out.print("x = " + x);
22      }
23    }
24 }
```

O programa acima segue o seguinte fluxo de execução, linha a linha: 3, 9, 10 (imprime `x = 10`), 11, 12 (imprime `x = 11`), 13 (exceção lançada), 16, 17 (imprime `x = 12`), 20, 3, 3, 21 (imprime `x = 13`), 22 e 4 (imprime `Acabou`).

Em princípio, a claúsula `finally` é sempre executada. Por exemplo, o `finally` é **sempre** executado, até mesmo quando o bloco do `try` ou `catch` executam `return`. Antes do retorno acontecer, o `finally` é executado, conforme mostra o programa, que imprime `x = 10  x = 11  x = 13  Acabou`.

```
1  public class Finally {
2     static public void main(String[ ] args) {
3       Final a = new Final( ); a.f(); System.out.print("Acabou");
4     }
5  }
6  class Final {
7     public void f( ) {
8        int x = 10;
9        System.out.print("x = " + x);
10       try { x = 11; System.out.print("x = " + x); return; }
11       catch(Exception e){ x = 12; System.out.print("x = " + x); }
```

```
12        finally { x = 13; System.out.print("x = " + x); }
13        System.out.print("Aqui não passa");
14     }
15  }
```

O mesmo vale para comandos `return` que ocorrem dentro de `catch`, os quais têm a mesma semântica dos que ocorrem dentro do corpo do `try`: somente são concluídos após a execução da cláusula `finally`. O programa abaixo imprime `x = 11`.

```
1  public class Finally {
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     public int f( ) {
9       int x = 0;
10      try { x = 1; throw new Exception( ); }
11      catch(Exception e) { x = x + 10; return x; }
12      finally { x = x + 100; }
13    }
14 }
```

Entretanto, alguns tipos de comandos geram um comportamento diferenciado e requerem que alguns cuidados devem ser tomados para garantir a semântica desejada. Por exemplo, a cláusula `finally` não é executada no caso de o bloco de `try` invocar a operação `System.exit`, para encerrar o programa. O programa abaixo imprime `x = 10` e não `x = 11`, demonstrando este fato:

```
1  public class Finally {
2     static public void main(String[ ] args){
3        Final a = new Final( ); a.f(); System.out.print("Acabou");
4     }
5  }
6  class Final {
7     static public void f( ) {
8        int x = 10;
9        System.out.println("x = " + x);
10       try { x = 11; System.exit(0); }
11       catch(Exception e){
12          x = 12;
13          System.out.println("x = " + x);
14       }
15       finally {
16          x = 11;
17          System.out.print("x = " + x);
18       }
19    }
20 }
```

Ressalte-se que a expressão `exp` que dá o valor de retorno de um comando `return exp` localizado dentro de um `try` é avaliada antes de o bloco da cláusula `finally` ser executado. Observe que o programa abaixo imprime `x = 10`, apesar de o valor final de `x` ser `1000`.

```
1  public class Finally {
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; return x; }
12      finally { x = 1000; }
13    }
14 }
```

O mesmo ocorre quando o `return` ocorre dentro de `catch`. O programa abaixo imprime `x = 107, a.y = 1100`:

```
1  public class Finally {
2     static public void main(String[ ] args){
3       Final a = new Final( ); int x = a.f();
4       System.out.println("x = " + x + ", a.y = " + a.y);
5     }
6  }
7  class Final {
8     public int y = 0;
9     public int f( ) {
10       int x = 0;
11       try { x = 1; throw new Exception( ); }
12       catch(Exception e) { x = 100; return x+7; }
13       finally { x = x + 1000; y = x; }
14    }
15 }
```

Comando `return` que ocorre na cláusula `finally` prevale sobre o do corpo do `try` ou de `catch`, sendo obedecido imediatamente. Por exemplo, no programa a seguir, o valor impresso é `x = 1000` e não `x = 10`.

```
1  public class Finally {
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 }
```

5.4 Objeto de Exceção

A classe de exceção é uma classe como qualquer outra, exceto pelo fato de pertencer a família `Throwable`. Uma classe de exceção pode conter livremente métodos e campos. 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, declaram-se uma classe de exceção para cada categoria de erro possível no programa:

```
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 }
```

Nessas classes, o atributo `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.

O tipo abstrato de dados `PilhaDeInteiros` é definido por:

```
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  }
```

Observe que as exceções `Tamanho`, `Máximo` e `Mínimo` são lançadas pela construtora de `PilhaDeInteiros`, operações `empilhe` e `desempilhe`, respectivamente. E todos os objetos lançados registram internamente um código, no caso o inteiro `1`, que informa que a origem dos lançamentos é a pilha.

O tipo abstrato de dados `FilaDeInteiros` é definido por:

```
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;
13       item[fim]= v;
14    }
15    public int remova( ) throws Mínimo {
16       if(vazia()) throw new Mínimo(2);
17       frente = (++inicio % tamanho);
18       return item[inicio];
19    }
20 }
```

As mesmas exceções `Tamanho`, `Máximo` e `Mínimo` são também lançadas pela construtora de `FilaDeInteiros`, operações `empilhe` e `desempilhe`, respectivamente. Mas agora o código de identificação da origem dos lançamentos é o `2`.

A inspecção de campos dos objetos de exceção passados aos `catch`es permite tomada de decisões em relação ao tipo de falha anotada na exceção. Por exemplo o teste da linha 11 informa se a exceção capturada foi lançada da pilha ou se foi da fila:

```
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 {
7             ... s.empilhe(i); ... k = s.desempilhe(); ...
8             ... q.insira(i);  ... k = q.remova();     ...
```

```
9          }
10         catch(Máximo e) {
11            if (e.id == 1)
12                 System.out.println("Foi na Pilha");
13            else System.out.println("Foi na Fila");
14         }
15         catch (Mínimo e) {
16            System.out.println("Foi na Pilha ou Fila");
17         }
18      }
19   }
20 }
```

5.5 Hierarquia de Exceções

Pode-se criar uma hierarquia de classes de exceção para melhor estruturar seu tratamento. A hierarquia permite agrupar o tratamento de erros quando isto for conveniente.

A hierarquia `Aritmético`, definida a seguir, agrupa as exceções de `Máximo`, de estouro de valor máximo, de `Mínimo`, de estouro de valor mínimo, e `Zero`, de divisão por zero.

```
1 class Aritmético extends Exception { }
2 class Máximo extends Aritmético { }
3 class Mínimo extends Aritmético { }
4 class Zero extends Aritmético { }
```

O usuário pode escolher tratar todas as exceções de uma só forma:

```
1 public class Usuário1 {
2    public static void main(String[ ] args) {
3       try { ... throw new Máximo( ); ... throw new Zero( ); ...
4             ... throw new Mínimo( ); ... throw new E( ); ...
5       }
6       catch(Aritmético  e) {trata Aritmético e descendentes }
7       ...
8 }
```

Ou então individualmente:

```
1 public class Usuário2 {
2    public static void main(String[ ] args) {
3       try { ... throw new Máximo( ); ... throw new Zero( ); ...
4             ... throw new Mínimo( ); ... throw new E( ); ...
5       }
6       catch(Máximo e) {trata Máximo e descendentes }
7       catch(Mínimo e) {trata Mínimo e descendentes }
8       catch(Zero e)   {trata Zero e descendentes }
9       ...
10 }
```

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. Isto porque a busca pelo `catch` é feita de cima para baixo, sendo abandonada quando for encontrado o primeiro que satisfizer o teste **é-um** relativo ao tipo do seu parâmetro. A ordem é automaticamente verificada pelo compilador, que, por exemplo, não aceita a ordenação de `catch`es usada no método `errado` abaixo, porque o primeiro `catch` iria capturar todas as exceções de sua hierarquia, e, consequentemente, as cláusulas seguintes 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 }
```

5.6 Informações Agregadas

Todo objeto de uma classe que estende `Throwable` ou `Exception` tem um campo contendo uma cadeia de caracteres que identifica a exceção. Este campo é usado com valor de retorno da operação `toString` associada ao objeto. Lembre-se que, em Java, se um objeto for usado onde uma cadeia de caracteres é esperada, a operação `toString` é automaticamente chamada. Este recurso é útil no processo de depuração do programa: a impressão de uma exceção que não deveria ocorrer ajuda a localizar o erro.

O uso dessa propriedade dos objetos de exceção é ilustrada pelo programa:

```
1  public class Agregado1 {
2    public static void main(String[] args) {
3       try { throw new ArrayStoreException(); }
4       catch (ArrayStoreException e) { System.out.println(e); }
5       try { throw new ExceçãoDoUsuário(); }
6       catch (ArrayStoreException e) {
7          String s = e.toString( );
8          System.out.println(e + "\n" + s);
9       }
10   }
11 }
12 class ExceçãoDoUsuário extends ArrayStoreException { }
```

que imprime:

```
java.lang.ArrayStoreException
ExceçãoDoUsuário
ExceçãodoUsuário
```

Informações podem ser acrescentadas à descrição da exceção. Para isto basta incluir na construtora da classe de exceção uma chamada à construtora de sua superclasse `Exception` que tem um parâmetro do tipo `String`, no momento da criação do objeto de exceção. Esta 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 }
```

5.7 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 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`.

5.8 Notas Bibliográficas

Capítulo 6

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. 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 que, desta forma, leitor iniciante na linguagem Java rapidamente se capacite 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.

As classes da biblioteca `java.lang` são automaticamente importadas para todo programa Java, podendo ser usadas livremente. As classes apresentadas neste capítulo são:

- Cálculos matemáticos:
  `Math`
- Manipulação de sequências de caracteres:
  `String`, `StringBuffer` e `StringTokenizer`
- Classes invólucro:
  `Integer`, `Boolean`, `Character`, `Byte`, `Float`, `Long` e `Double`

As classes invólucro são usadas para dar status de objetos a valores de tipos primitivos.

6.1 Cálculos Matemáticos

A classe `Math` possui métodos para realizar cálculos matemáticos comuns, como raiz quadrada, logaritmo e de trigonometria. A biblioteca `java.lang` provê a classe `Math`, que define constantes e funções matemáticas. Os membros de Math são implementados como estáticos, podendo ser ativados sem a necessidade de se criar objetos da classe.

Os argumentos das funções matemáticas apresentadas na tabela a seguir são todos do tipo `double`, e os seus valores devem ser fornecidos em **radianos**. Todas as funções, exceto `round`, `ceil` e `floor`, retornam um valor do tipo `double`.

Função	Valor Retornado
`Math.PI`	constante π (aprox. 3.141592653589793)
`Math.E`	constante 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`, desempate usa inteiro par
`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`

6.2 Literal `String`

A classe `String` implementa operações para manipulação de sequências de caracteres. Diferentemente de classes definidas pelo programador, `String` possui notação especial para suas constantes, também chamadas de literais, que são codificadas como seqüências de caracteres do conjunto UNICODE colocadas entre `"`. 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 deste objeto.

Expressões contendo somente literais `String` são avaliadas em tempo de compilação, e literais idênticos produzem objetos de mesma referência, isto é, o mesmo objeto. Por exemplo, a expressão `"uma parte " + "outra parte"` gera exatamente o mesmo objeto que a sequência `"uma parte outra parte"`. Se ambos os literais ocorrerem em

um programa, mesmo em módulos compilados separadamente, um único objeto `String` será gerado 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 unificação de literais citada acima somente ocorre durante a compilação.

A função `main` da classe `ConstanteString` abaixo cria três objetos do tipo `String`: dois durante compilação, 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 tem iniciada sua execução, os objetos mostrados na Figura 6.1 são imediatamente alocados.

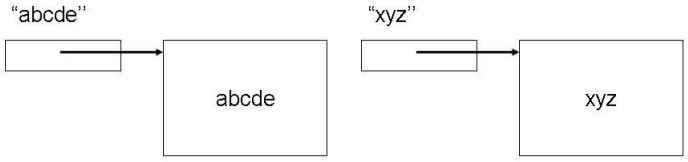


Figura 6.1 Constante String I

A declaração da linha 3 acrescenta a referência `s1`, como mostrado na Figura 6.2.

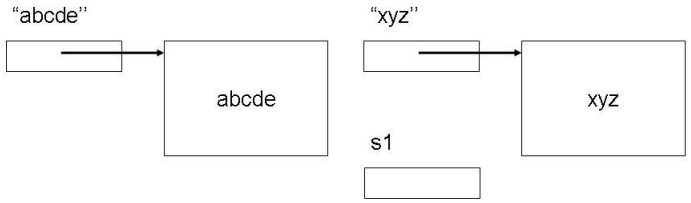


Figura 6.2 Constante String II

A atribuição da linha 4 cria o compartilhamento de objetos indicado na Figura 6.3. O valor impresso pelo comando na linha 5 é: `abcdexyzxyz`.

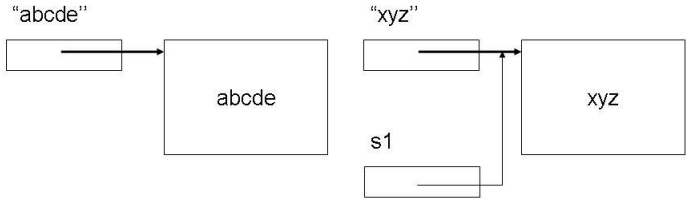


Figura 6.3 Constante String III

6.3 Classe `String`

A classe `String` não possui operações para modificar o estado do objeto corrente. Por isto, objetos do tipo `String` são imutáveis 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 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 6.4.

Todo objeto em Java tem uma operação com a assinatura `String toString( )`, que produz alguma informação textual sobre o objeto. Este recurso é útil, por exemplo, no rastreamento da execução de um programa. Para facilitar o tratamento desses objetos como se fossem do tipo `String`, Java realiza conversão implícita, sempre que necessário, de qualquer objeto para `String`, inserindo adequadamente no código a operação `toString`. A operação `toString` default pode ser redefinida pelo programador. Por exemplo, o comando a linha `A`.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 **http://www.javasoft.com**, mantida pelos criadores da linguagem.

As operações de `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 esta 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 referências para `String` podem ser iniciados diretamente em sua declaração, omitindo-se 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, como mostra a Figura 6.4, que retrata a situação após a execução dos comandos da linha 5 de `ArranjosDeString1`.

```
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  }
```

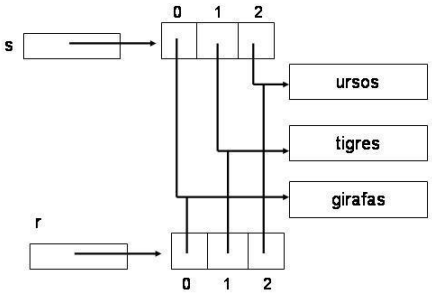


Figura 6.4 Arranjos de Strings

Note, na Figura 6.4, que os objetos contendo os literais `String` de mesmo valor são compartilhados.

O uso do operador new para alocação de arranjo iniciado na declaração pode ser explicitamente especificado, caso o programador assim o desejar. O programa `ArranjoDeString2` é equivalente a `ArranjoDeString1`:

```
1  public class ArranjosDeString2 {
2     public static void main (String[ ] args) {
3        String[ ] s = new String[ ] {"girafas", "tigres", "ursos"};
4        String[ ] r = new String[3];
5        r[0] = "girafas"; r[1] = "tigres"; r[2] = "ursos";
6     }
7  }
```

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 de `t`.

  ```
  String s = new String("abcde");
  ```

- `public String(char v[])`: constrói um 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.

Tamanho de sequências de Caracteres

O tamanho da sequência de caracteres armazenada em um objeto `String` é dado pela operação:

- `public int length()`: retorna o número de caracteres no `String` **this**.

```
String s = "abcde";
int n = s.length();  // n recebe 5
```

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 do objeto corrente. O valor de `p` que deve estar no intervalo $[0, \texttt{this.length}() - 1]$. O valor `0` denota a primeira posição, o valor `1`, a segunda, e assim por diante. Um `p` fora do intervalo causa o levantamento da exceção `IndexOutOfBoundsException`.

```
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` `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 8 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, noturna sombra";
4        char c;
5        int [] contagem = new int[256];
6        for (int i=0; i < s.length(); i++) {
7            c = s.charAt(i);
8            if (c < 256) contagem[c]++;
9        }
10    }
11 }
```

Comparação de Objetos `String`

sequências de caracteres são representadas como objetos em Java. Variáveis do tipo `String` são referências para objeto. 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.

A operação `s1 == s2`, onde `s1` e `s2` são referências para objetos `String`, apenas verifica se `s1` e `s2` apontam para o mesmo objeto `String`. Por outro lado, 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 de **this** tiver o mesmo conteúdo que o do `String` representado pelo objeto `x`.
  `boolean b = s1.equals(s2);`

- `public boolean equalsIgnoreCase(Object x)`: retorna `true` se a sequência de caracteres de **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 de **this** seja lexicograficamente menor, igual ou maior que a de `s`, respectivamente.
  `int k = s1.compareTo(s2);`

O programa `TabelaSequencial` ilustra o uso da operação `equals` com a implementação de um novo tipo com as operações `inclui` e `pesquisa`. O tipo é representado por um arranjo não ordenado de tamanho 100 de `String` e não se permite elementos repetidos. As construções `throws` da linha 5 e `throw` da linha 8 cuidam da situação de erro causada por falta de espaço na tabela para inserção de novos elementos. O funcionamento do tratamento de falhas está detalhado no Capítulo 5.

```
1  public class TabelaSequencial {
2     private final static int MAX = 100;
3     private String[] tabela = new String[MAX];
4     private int tamanho = 0;
5     public int inclui(String chave) throws Estouro {
6        for (int i=0; i < tamanho; i++)
7            if (tabela[i].equals(chave)) return i;
8        if (tamanho == MAX ) throw new Estouro();
9        int novo = tamanho++;
10       tabela[novo] = chave;
11       return novo;
12    }
13   public int pesquise(String chave) {
14       for (int i=0; i < tamanho; i++)
15           if (tabela[i].equals(chave)) return i;
16       return -1;
17   }
18 }
```

A operação `a.equals(b)` requer que `a` e `b` sejam referências para objetos do tipo `String` e serve para verificar a igualdade das sequências de caracteres contidas nos objetos apontados por `a` e `b`. O parâmetro `b` pode ser uma referência nula. Neste caso, a função `equals` retorna sempre `false`, para qualquer que seja o objeto apontado por `a`, que nunca pode ser nulo. Observe que o programa `sequênciaReferência` abaixo não contém erro de endereçamento:

```
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 }
```

e produz a saída: `Funcionou e são diferentes`

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 toda sequência de caracteres é devidamente alocada pelo compilador.

Essa vantagem é destacada no próximo programa, que tem um erro de referência somente detectável durante a execução. Claramente a construção da linha 4 é superior à da linha 7, porque esta causa o aborto da execução com a mensagem `Exception in thread "main"java.lang.NullPointerException`, pela tentativa de acesso a um objeto via uma referência nula, enquanto que a da linha 4 apenas informa que as sequências representadas por `x` e `"abc"` são diferentes.

```
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 implementação de `TabelaBinária` a seguir ilustra o uso da operação `compareTo` da classe `String`. Nesta implementação, a tabela de chaves é mantida sempre ordenada de forma a possibilitar o uso da técnica de pesquisa binária.

A inserção de uma chave inicia-se pela pesquisa na tabela pela chave que se deseja inserir. Caso a encontre, apenas retorna-se o índice da chave na tabela. Caso contrário, a nova 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 `TabelaBinária`.23, que calcula a posição do meio da porção da tabela sendo pesquisada, é equivalente a `(inf + sup)/2`, exceto que esta fórmula está sujeita a *overflow*, quando `sup` tiver valores próximos ao do maior `int` permitido na linguagem.

```
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)
13                tabela[i] = tabela[i-1];
14           else { novo = i ; break; }
15       }
16       tabela[novo] = chave;
17       return novo;
18    }
19    public int pesquise(String chave) {
20       int inf = 0, sup = tamanho - 1;
21       int meio, c;
22       while (inf <= sup) {
23          int meio = inf + (sup - inf)/2;
24          int c = chave.compareTo(tabela[meio]);
25          if (c == 0) return meio;
26          else if (c < 0) sup = meio - 1; else inf = meio + 1;
27       }
28       return -1;
29    }
30 }
```

Área de Objetos Canônicos

Comparar somente referências para objetos `String` é mais eficiente que comparar os conteúdos armazenados nos objetos 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 privativa de objetos canônicos do tipo `String`. Esta área pode ser administrada com o auxílio do método `public String intern()`.

Quando o método `intern` for chamado para um objeto receptor `s`, se a área de objetos canônicos privativos já possuir um objeto contendo exatamente a mesma sequência de caracteres representada por `s`, segundo a operação `equals`, o endereço do 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 da 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`, que mostra que a função `intern` retorna o mesmo objeto canônico para as ocorrências do objeto literal `"abcde"` e os objetos criados pelas operações `"abc" + "de"` da linha 6 e `("abc" + de).intern()` da linha 8.

Localização de Caracteres

As operações para localizar ocorrência de um caractere dentro da sequência de caracteres de um objeto `String` são:

- `public int indexOf(char c)`: retorna a posição (≥ 0) da primeira ocorrência do caractere `c` na sequência do **this**. Retorna `-1` se não achar.

  ```
  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
  ```

O método `Localiza.numCharacteresEntre` retorna o número de caracteres localizados entre a primeira e a última ocorrências de um dado caractere `c` em um `String s`, passados com parâmetros. Caso o caractere não seja encontrado, retorna-se `-1`:

```
1  public class Localiza {
2     static int numCaracteresEntre(String s, char c) {
3        int inicio = s.indexOf(c);
4        if (inicio < 0) return 0;
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`.

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 no objeto `s`.

```
String s1 = new String("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 = new String("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` no string **this**.

```
String s1 = new String("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 = new String("abc-dfg-hjk-dfg-gmn-opq-dfg-mhj-klm");
String s2 = new String("dfg");
int k = s1.indexOf(s2, 15); // k recebe 12
```

Particionamento de sequências

A operação `split` da classe `String` particiona a sequência de caracteres contidas no objeto corrente em subsequências, com base em um conjunto de delimitadores. A 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 é 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 esta sequência é o delimitador das subsequências.

Entretanto, padrões mais gerais podem ser definidos. É comum expressões regulares que denotam um conjunto de delimitadores. Por exemplo, o padrão `"[:=*]"` especifica que o delimitador de subsequências são `:`, `=` ou `*`. Pode haver padrões que são combinações de subpadrões.

Alguns dos padrões permitidos estão exemplificados na tabela:

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 (complementação)
`[a-zA-Z]`	caractere de a até z ou A até Z, inclusive os extremos
`[a-d[m-p]]`	caractere de a até d, or m até p: [a-dm-p] (união)
`[a-z&&[def]]`	caractere de d, e, or f (interseção)
`[a-z&&[^bc]]`	caractere de a até z, exceto b e c: [ad-z] (diferença)
`[a-z&&[^m-p]]`	caractere de a até z, excluindo de m até p: [a-lq-z](diferença)

Os programas `Particiona*`, 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 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 do programa é: `acc-xyz-dccz-  Tamanho de t = 3`

O delimitador não é incluído nas subsequências retornadas. Neste exemplo, a primeira partição vai do início da sequência até o caractere que antecede a primeira ocorrência do 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"); // u = {"a", "", ":xyz:d", "", "z"}
5       for (String v : u) System.out.print(v + "-");
6       System.out.println("  Tamanho de u = " + u.length);
7    }
8 }
```

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 do 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 (int i = 0; i < a.length; i++ )
7            System.out.print(a[i] + "-");
8    }
9 }
```

Neste caso, a saída é : `abc-cde-efg-hij`

Construção de sequências de Caracteres

As seguintes operações são úteis para a construção de novas sequências de caracteres a partir de sequências dadas:

- `public String concat(String s)`: forma uma nova sequência pela concatenação de uma cópia da sequência do objeto corrente com uma 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"
```

6.4 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. Toda alteração em uma sequência gera um novo objeto. Esta restrição é útil para garantir que objetos do `String` compartilhados não serão alterados.

Para implementar objetos contendo sequências de caracteres que podem ser alteradas *in loco*, deve-se usar a classe `StringBuffer` em vez de `String`. Objetos da classe `StringBuffer` armazenam internamente uma sequência de caracteres, chamada de **sequência buferizada**, cujo comprimento e conteúdo podem ser mudados via chamadas de certos métodos definidos na classe `StringBuffer`.

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"`, acrescentada no seu fim. O objeto resultante, do tipo `StringBuffer`, é convertido para `String` via a operação `toString`.

As principais operações de `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 buferizada do objeto `s` para `s = "abcdedfxy"`, enquanto a chamada `s.insert(4,"xy")` teria atualizado a mesma sequência para `s = "abcdxyefg"`.

Todo objeto `StringBuffer` é criado com uma capacidade máxima pré-definida para representar sua sequência buferizada. Caso esta capacidade do objeto torne-se insuficiente, a área interna da sequência é automaticamente aumentada. A posição inicial de uma sequência buferizada é `0`.

Construtoras de `StringBuffer`

São três as principais funções construtoras da classe `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`.

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 à direita o conteúdo original uma posição a partir de `p`. Lança-se a exceção `IndexOutOfBoundsException`, se `p` não estiver no intervalo $[0, \texttt{this.length}()]$.
- `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. Levanta-se `IndexOutOfBoundsException`, se `p` não estiver no intervalo $[0, \texttt{this.length}()]$.
- `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 `IndexOutOfBoundsException`, se `p` não estiver no intervalo $[0, \texttt{this.length}()]$.

Controle de Capacidade

As sequências buferizadas são criadas com um tamanho definido pelo usuário ou pelo valor default de 16. Este 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 de `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 armazenados na sequência buferizada de `this`.

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. A exceção `StringIndexOutOfBoundsException` é levantada se `início` estiver fora do intervalo [`0`, `this.length`()]. O método retorna o valor de `this`.
- `StringBuffer deleteCharAt(int p)`: remove o caractere da posição `p` da sequência buferizada, compactando o resultado. Se o parâmetro `p` estiver fora do intervalo [`0`, `this.length`()], lança-se a 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 desta sequência tiver 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.

Recuperação de Subsequências

As seguintes operações de `StringBuffer` servem para obter um ou mais dos caracteres armazenados em um objeto desta 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`()].

- `String substring(int início)`: retorna uma subsequência com a sequência de caracteres correntemente armazenados na sequência buferizada, desde a posição `início` até o seu fim. Levanta-se a exceção `IndexOutOfBoundsException`, se `início` não estiver no intervalo [0, `this.length`()].

- `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 `início` ou `fim` não estiverem no intervalo [0, `this.length`()].

- `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 no intervalo [0, `this.length`()]; (iii) `p+fim-início > d.length()`.

Substituição de Caracteres

- `StringBuffer replace(int início, int fim, String s)`: substitui os caracteres da subsequência que ocupam as posições de `início` 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 `início` 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`.

Uso de `StringBuffer`

O programa `Substituição` troca todas as ocorrências do caractere ´`a`´ da sequência do objeto `StringBuffer assis` 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        corrige(assis,'a', '*');
6        System.out.println(assis);
7     }
8     public static void corrige(StringBuffer s, char v, char n) {
9        for (int i = 0; i < s.length(); i++)
10            if (s.charAt(i) == v) s.setChar(i,n);
```

```
11     }
12  }
```

A saída é: `Não se me daria olhar ==> Não se me d*ri* olh*r`

O programa `Explicação` monta uma sequência de caracteres explicando a raiz quadrada de um número. A capacidade do `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(").append(i).append(')');
9        b.append(" = ").append(Math.ceil(Math.sqrt(i)));
10       return b.toString();
11    }
12 }
```

A saída é: `A parte inteira de sqrt(256) = 16`

O programa `DataDeHoje` acrescenta um texto no fim de uma sequência de caracteres. A operação realizada pelo método `adiciona` equivale a uma concatenação, isto é, o programa faz `t = s + a`. A execução do método `ensureCapacity` da linha 12 dá a garantia de que, durante a execução da linha 13, a sequência buferizada de `c` não terá que 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 2007";
5        String t = adiciona(s,a); // t = s + a
6        System.out.println(t);
7     }
8     public static String adiciona(String b, String a) {
9        StringBuffer c = new StringBuffer(c);
10       int n1   = c.length();
11       int n2   = a.length();
12       c.ensureCapacity(n1 + n2 + 2);
13       c.insert(n1," ").insert(n1+1,a).insert(n1+n2+1,".");
14       return c.toString();
15    }
16 }
```

O programa produz a saída: `Hoje é dia 15 de outubro de 2007.`

6.5 Classe `StringTokenizer`

A classe `StringTokenizer` serve para particionar uma sequência de caracteres em *tokens*. Tipicamente um *token* é um delimitador ou a maior subsequência de caracteres que não contém delimitadores. Os delimitadores reconhecidos no processo de

particionamento são definidos por *default* ou podem ser especificados no momento da criação do objeto `StringTokenizer`. Pode-se também controlar se delimitadores devem ser tratados como *tokens*. Por default, delimitadores não são *tokens* e são pré-definidos pelo padrão `" \t\n\r\f"`, que os especifica como sendo os caracteres branco, de tabulação, de mudança de linha, de retorno de carro e de avanço de *formulário*.

No exemplo a seguir, os delimitadores são os definidos por *default*. Desta forma, o efeito do programa abaixo é separar as palavras da sequência de caracteres dada, tomando espaço em branco como delimitador de *tokens*.

```
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-`.

Construtoras de `StringTokenizer`

As funções construtoras de `StringTokenizer` são:

- `public StringTokenizer(String s)`: constrói um objeto `StringTokenizer` para a sequência de caracteres `s`, usando os delimitadores *default*, que não são tratados como *tokens*.
- `StringTokenizer(String s, String d)`: constrói um objeto `StringTokenizer` para a sequência de caracteres `s`, usando como delimitadores de *tokens* os caracteres contidos na sequência `d`. Os delimitadores não são tratados como *tokens*.
- `public StringTokenizer(String s, String d, boolean dToken)`: constrói o objeto `StringTokenizer` para a sequência `s`, usando como delimitadores de *tokens* os caracteres em `d`. O parâmetro `dToken` quando igual a `true` indica que delimitadores devem ser tratados como *tokens*. Se `false`, delimitadores são apenas separadores. Se `d` for igual a `null`, a invocação de operações do objeto criado causa o lançamento da exceção `NullPointerException`.

As funções construtoras de `StringTokenizer` sempre iniciam a posição corrente do próximo *token* para o início do primeiro *token* na sequência de caracteres do objeto corrente. Esta posição corrente é usada pelas operações a seguir.

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. Esta 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* reconhcido 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.

6.6 Classes de Tipos Primitivos

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 foi declarado. 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 a referência 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 Primitivo	Classe Invólucro	Tipo Primitivo	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 tipos primitivos como objetos e operações para desencaixotá-los de volta. O programa `Invólucro` abaixo mostra como valores de tipos primitivos podem ser encaixotados e desencaixotados:

```
1 public class Invólucro {
```

```
2     int i, j = 50;
3     Integer k = new Integer(j); // encaixotamento em k do inteiro j
4     i = k.intValue();           // desencaixotamento do inteiro em k
5     float x, y = 100.0;
6     Float z = new Float(y);     // encaixotamento em z do float y
7     x = z.FloatValue();         // desencaixotamento do float em z
8     System.out.print("i= " +i+ " j= " +j + " x= " +x+ " y= " +y);
9  }
```

A saída do programa é : `i= 50 j= 50 x= 100.0 y= 100.0`

As classes invólucro possuem, além das operações de encaixotamento e desencaixotamento 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 http://www.javasoft.com.

Classe `Boolean`

Objetos do tipo `Boolean` encaixotam valores do tipo `boolean`. Suas operações principais são:

- `public Boolean(boolean value)`
- `public String toString()`
- `public boolean booleanValue()`

A função `toString` retorna a sequência `"TRUE"` ou `"FALSE"`, conforme seja o valor encaixotado no objeto corrente.

Classe `Character`

Objetos do tipo `Character` representam valores primitivos do tipo `char`. Principais operações são:

- `public Character(char value)`
- `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.

Classe `Number`

`Number` é uma classe abstrata que tem as subclasses `Integer`, `Long`, `Float` e `Double`, as quais encaixotam os respectivos tipos primitivos e definem métodos para fornecer o valor numérico encaixotado pelo tipo correspondente, isto é, para `int`, `long`, `float` ou `double`.

- `public abstract int intValue()`
- `public abstract long longValue()`
- `public abstract float floatValue()`
- `public abstract double doubleValue()`

Classe `Integer`

Subclasse de `Number` com as operações:

- `public Integer(int value)`
- `public String toString()`
- `public int intValue()`
- `public static int parseInt(String s) throws NumberFormatException`

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.

Classe `Long`

Subclasse de `Number` com as operações:

- `public Long(long value)`
- `public Long(String s) throws NumberFormatException`
- `public long longValue()`
- `public static long parseLong(String s) throws NumberFormatException`

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.

Classe `Float`

Subclasse de `Number` com as operações:

- `public Float(float value)`
- `public String toString()`
- `public float floatValue()`

Classe `Double`

Subclasse de `Number` com as operações:

- `public Double(double value)`
- `public double doubleValue()`

6.7 Exercícios

1. Quando deve-se usar `String` e quando deve-se usar `StringBuffer`?

2. Qual é o tipo de um literal formado por uma sequência de caracteres?

3. Por que classes invólucros são necessárias?

6.8 Notas Bibliográficas

Capítulo 7

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 se controlar a visibilidade de tipos ao longo do programa. Em particular, ele permite fazer um tipo valer só 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 é o resultado da combinação de estrutura de bloco com programação baseada em classes. Tipos aninhados em classes podem ter os mesmos modificadores de acesso (`public`, `protected`, `private` ou *pacote*) que membros normais.

A compilação de arquivo-fonte com classes que contenham classes 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 de nome `NomeClasseExterna$NomeClasseInterna.class`, e os das classes externas, em `NomeClasseExterna.class`.

Quanto à capacidade de ter acesso a elementos do escopo envolvente, as classes aninhadas podem ser **estáticas** ou **não-estáticas**. Interfaces aninhadas são sempre estáticas.

7.1 Classes Aninhadas Estáticas

Uma classe aninhada estática é como uma classe externa, exceto que seu nome e sua acessibilidade externa é definida pelo seu modificador de acesso. Uma classe aninhada estática somente pode fazer referências a membros estáticos da 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, podeSacar, podeEncerrar;
5        void verifiquePermissão(Conta c) {
6           ... c.número ...     // OK
7           ... this.número ...  // Erro
8        }
9     }
10    public Permissões obtémPermissão(){ return new Permissões( ); }
11 }
12 class Banco {
13    Conta c = new Conta( );
14    Conta.Permissões p1 = new Conta.Permissões( );
15    void f() {
16       Conta.Permissões p2;
17       p2 = c.obtémPermissão(  );
18       if  (p1.podeDepositar && p2.podeSacar) ....
19    }
20 }
```

Referências a tipo aninhado estático fora da classe que o declara devem ser qualificadas pelo nome da classe envolvente, como `Conta.Permissões` da linha `Banco`.14. A classe aninhada pode ser usada localmente sem qualificações, como ocorre na linha `Conta`.10. Os atributos da classe envolvente `Conta` declarados na linha `Conta.??` não são acessíveis no corpo de classe estática `Permissões`: a tentativa de acesso mostrada na linha `Conta`.7 provoca erro de compilação. Os atributos públicos de `Permissões` são acessíveis em todo local que `Conta.Permissões` o for, como ocorre na linha 18. 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*. Por exemplo, o seguinte programa funciona e imprime `y = 3000`.

```
1  public class A {
2     static class B {
3         private int x = 1000;
4         public  int z = 2000;
5     }
6     static public void main(String[] args) {
7        B b = new B();
8        int y = b.z;                  // OK
9        y += b.x;                     // OK, embora  b.x seja private
10       System.out.print("y = " + y);
11    }
12 }
```

A classe `A` a seguir tem erro de compilação devido à tentativa de acesso a membro privado fora de seu escopo, ocorrida na linha `A`.11:

```
1  class C {
2     public static class B {
3         private int x = 1000;
```

```
4         public  int z = 2000;
5     }
6  }
7  public class A {
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 }
```

7.2 Classes Aninhadas Não-Estáticas

Classe aninhada não-estática é um conceito distinto do de classe aninhada estática. Enquanto este constitui-se essencialmente em um mecanismo de controle de visibilidade e escopo de tipos, o primeiro, 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 estes privilégios de acesso de sua superclasse.

Classe aninhada não-estática cria a noção de **objetos envolventes** e **objetos envolvidos**.

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 pelas 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, podeSacar, podeEncerrar;
5        void verifiquePermissão(Conta c) {
6           ... c.número ...     // OK
7           ... this.número ...  // OK
8        }
9     }
10    public Permissões obtémPermissão( ) { return new Permissões( ); }
11 }
12 class Banco {
13    Conta c = new Conta( );
14    Conta.Permissões p1 = c.new Permissões(  );
15    void f() {
16       Conta.Permissões p2;
17       p2 = c.obtémPermissão( );
18       if  (p1.podeDepositar && p2.podeSacar) ...
19    }
20 }
```

Destaca-se que nesta nova versão de `Conta`, a referência ao atributo `número` da classe envolvente, ocorrida na linha `Conta`.7 está correta, e que, na linha `Conta`.14, o operador `new` apresenta-se com uma sintaxe e uma semântica especiais.

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. Se a referência ao objeto envolvente for omitido no operador `new`, toma-se por *default* a referência ao objeto que está criando a instância da classe aninhada, i.e., o `this`. O uso de qualificação no nome da construtora de um `new` de classe aninhada não-estática não é permitido:

```
c.new Conta.Permissões( );// Errado, após new somente identificador
new Conta.Permissões( );  // Errado, Permissões não é estática
c.new Permissões( );      // OK
```

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 outras de classes envolventes e que 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 mostra como se fazem referências não-ambíguas a membros das classes envolventes em um dado aninhamento:

```
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 = a de X
8           ... b ...          // Y.this.b = b de Y
9           ... c ...          // Z.this.c = c de Z
10          ... X.this.d ...   // d de X
11          ... Y.this.d ...   // d de Y
12          ... d ...          // Z.this.d = 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`, como na linha 12, é interpretado como `Z.this.d`, referindo-se ao `d` que está declarado mais próximo, dentro de `Z`.

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`. Mas 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` ocorrer dentro de um método de um objeto que foi criado pelo comando `y.new Z( )`, então `Y.this` é um campo do objeto `Z` criado e tem 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.

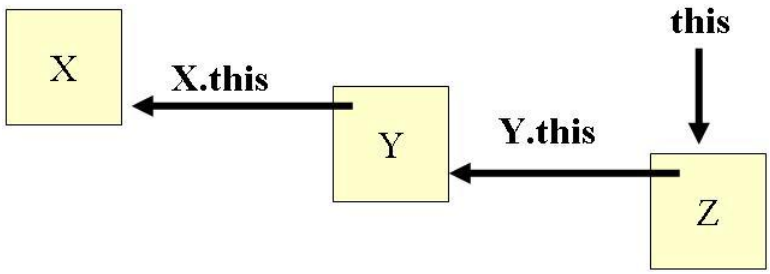


Figura 7.1 Objetos Envolventes

Figura 7.1 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    public static void main(String[ ] args) {
14       A x = new X ( );
15       X.Y y = x.new Y( );
16       X.Y.Z  z = y.new Z( );
17    }
18 }
```

7.3 Exemplo de Classes Aninhadas

O programa completo a seguir tem o objetivo de mostrar o encadeamento de objetos envolvidos e envolventes. A classe `X` possui uma classe interna `Y`, que por sua vez possui uma interna `Z`. Inicialmente criam-se dois objetos `X`, que por sua vez criam objetos de classes internas. Cada passo de execução é impresso para permitir seu acompanhamento.

```
1  class X {
2     int a;
3     PrintStream o = System.out;
4     public X(int a) { this.a = a;}
5     void criaY(String s, X x) {
6        o.println("    imprime a = " + a);
7        o.println("    cria y1 com b = 2 envolvido por " + s );
8        Y y1 = x.new Y(2);
9        o.println("    cria y2 com b = 200 e envolvido por " + s );
10       Y y2 = x.new Y(200) ;
11       o.println("y1.print, sendo y1 envolvido por " + s + ":");
12       y1.criaZ("y1", y1);
13       o.println("y2.print, sendo y2 envolvido por " + s + ":");
14       y2.criaZ("y2", y2);
15    }
16    public class Y {
17       int b;
18       public Y(int b) { this.b = b; }
19       void criaZ(String s, Y y) {
20          o.println("    imprime a = " + a);
21          o.println("    imprime X.this.a = " + X.this.a);
22          o.println("    imprime b = " + b );
23          o.println("    Cria z com c = 3 e envolvido por " + s );
24          Z z = y.new Z(3);
25          o.println("z.print, sendo z  envolvido por " + s + ":");
26          z.mostraReferências("z ");
27       }
28       public class Z {
29          int c;
30          public Z(int c) { this.c = c; }
31          void mostraReferências(String s) {
32             o.println("    imprime a = " + a);
33             o.println("    imprime X.this.a = " + X.this.a);
34             o.println("    imprime b = " + b );
35             o.println("    imprime c = " + c );
36          }
37       }
38    }
39 }
```

O programa abaixo cria dois objetos do tipo `X`, os quais criam objetos envolvidos das classes `Y` e `Z`:

```
1  public class TesteDeAninhamento {
2     public static void main(String[ ] args) {
3        PrintStream o = System.out;
4        o.println("Cria x1 com a = 1 ");
5        X x1 = new X(1);
6        X x2 = new X(100);
7        o.println("Cria x2 com a = 100 ");
8        o.println("x1.print:");
9        x1.criaY("x1", x1);
10       o.println("x2.print:");
11       x2.criaY("x2", x2);
12    }
13 }
```

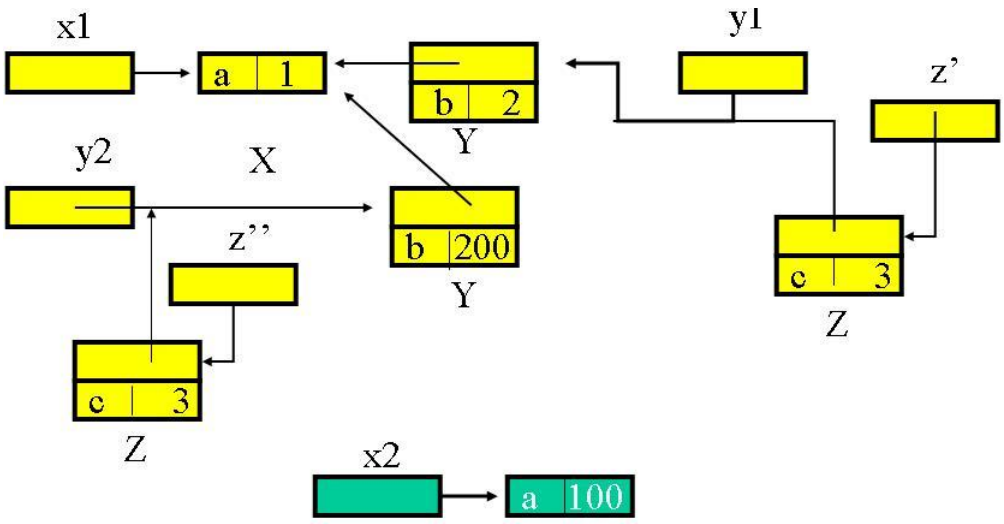


Figura 7.2 Encadeamento de Objetos Aninhados

Figura 7.2 mostra a estrutura dos objetos alocados pelo programa `TesteDeaninhamento`, cuja saída é:

```
Cria x1 com a = 1
Cria x2 com a = 100
x1.print:
    imprime a = 1
    cria y1 com b = 2 envolvido por x1
    cria y2 com b = 200 e envolvido por x1
y1.print, sendo y1 envolvido por x1:
    imprime a = 1
    imprime X.this.a = 1
    imprime b = 2
    Cria z com c = 3 e envolvido por y1
```

```
z.print, sendo z  envolvido por y1:
    imprime a = 1
    imprime X.this.a = 1
    imprime b = 2
    imprime c = 3
y2.print, sendo y2 envolvido por x1:
    imprime a = 1
    imprime X.this.a = 1
    imprime b = 200
    Cria z com c = 3 e envolvido por y2
z.print, sendo z  envolvido por y2:
    imprime a = 1
    imprime X.this.a = 1
    imprime b = 200
    imprime c = 3
x2.print:
    imprime a = 100
    cria y1 com b = 2 envolvido por x2
    cria y2 com b = 200 e envolvido por x2
y1.print, sendo y1 envolvido por x2:
    imprime a = 100
    imprime X.this.a = 100
    imprime b = 2
    Cria z com c = 3 e envolvido por y1
z.print, sendo z  envolvido por y1:
    imprime a = 100
    imprime X.this.a = 100
    imprime b = 2
    imprime c = 3
y2.print, sendo y2 envolvido por x2:
    imprime a = 100
    imprime X.this.a = 100
    imprime b = 200
    Cria z com c = 3 e envolvido por y2
z.print, sendo z  envolvido por y2:
    imprime a = 100
    imprime X.this.a = 100
    imprime b = 200
    imprime c = 3
```

7.4 Classes Locais em Blocos

Classes podem ser definidas dentro de blocos de código, particularmente dentro de métodos. Como são locais a blocos, tais como qualquer variável, não são acessíveis fora do bloco em que foram definidas. Essas classes têm uso puramente local, seus corpos têm acesso a certos nomes que estejam disponíveis no bloco e aos membros da classe que contém o bloco. A restrição é que permite-se acesso somente a variáveis do bloco

que sejam declaradas finais.

As instâncias de classes locais são objetos normais que podem ser retornados por funções, passados como parâmetros. O uso desses objetos fora do bloco onde suas classes foram declaradas é possível se essas classes implementarem interfaces conhecidas externamente como ilustra o exemplo a seguir, que imprime `0 1 2 3 4 5 6 7 8 9`:

```
1  interface Enumeração {
2     boolean aindaHá( );
3     Object próximo( ) throws FaltaElemento;
4     void reInicia( ) ;
5  }
6  class FaltaElemento extends Exception { }
7  class Aluno {
8     private int n;
9     Aluno(int n) { this.n = n; }
10    int matricula( ) { return n; }
11 }
12 class CorpoDiscente {
13    private Object[ ] objetos;
14    public CorpoDiscente(Object[] objetos)this.objetos = objetos;
15    public Enumeração elementos( ) {
16       final int tamanho = objetos.length;
17       class EnumeraçãoLocal implements Enumeração {
18          private int posição = 0;
19          public boolean aindaHá( ) {return (posição < tamanho); }
20          public Object próximo( ) throws FaltaElemento {
21             if (posição >= tamanho) throw new FaltaElemento();
22                 return objetos[posição++];
23          }
24          public void reInicia( ) { posição = 0; }
25       }
26       return new EnumeraçãoLocal( );
27    }
28 }
29 public class Acadêmico {
30    public static void main(String[ ] args) {
31      Aluno a, alunos[ ] = new Aluno[10];
32      for (int i = 0; i < alunos.length; i++) alunos[i] = new Aluno(i);
33      CorpoDiscente c = new CorpoDiscente(alunos);
34      Enumeração e = c.elementos( );
35      while (e.aindaHá( )) {
36         try {
37            a = (Aluno) e.próximo( );
38            System.out.print(a.matricula() + " ");
39         }
40         catch(FaltaElemento x) { }
41      }
```

```
42      }
43  }
```

Ao retornar da invocação de `c.elementos` da linha `CorpoDiscente`.34, o valor da variável local `tamanho` (linha `CorpoDiscente`.16), que foi alocada no momento de entrada do método, não pode ser perdido, porque, ao se criar um objeto da classe local `EnumeraçãoLocal`, cria-se uma entidade que faz, nas linhas `CorpoDiscente`.19 e `CorpoDiscente`.21, referências à variável local `tamanho` e que sobrevive à execução de `elementos`. Variáveis locais, como `tamanho`, que são referidas por objetos de classe interna a método devem ser declaradas `final`, para garantir que os seus valores sobrevivem 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 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.

7.5 Classes Anônimas

Se uma classe for declarada para ser usada uma única vez no próprio local onde foi declarada, seu nome pode ser omitido, declarando-se a classe diretamente como operando do operador `new`. Por exemplo, considere a classe `CorpoDiscente` do exemplo anterior, que contém a classe `EnumeraçãoLocal`, declarada dentro do método `elementos`:

```
1  class CorpoDiscente {
2     private final Object[ ] objetos;
3     public CorpoDiscente(Object[] objetos)this.objetos = objetos;
4     public Enumeração elementos( ) {
5        final int tamanho = objetos.length;
6        class EnumeraçãoLocal implements Enumeração {
7           private int posição = 0;
8           public boolean aindaHá( ) {return (posição < tamanho); }
9           public Object próximo( ) throws FaltaElemento {
10             if (posição >= tamanho) throw new FaltaElemento();
11                 return objetos[posição++];
12          }
13          public void reInicia( ) { posição = 0; }
14       }
15       return new EnumeraçãoLocal( );
16    }
17 }
```

A classe local `EnumeraçãoLocal` tem garantidamente um único uso, que é o da linha `CorpoDiscente`.15. Seu nome adiciona muito pouco à clareza e pode ser eliminado do código, substituindo a construtora do `new` por uma declaração anônima, como mostra a nova versão de `CorpoDiscente`:

```
1  public Enumeração elementos( ) {
2     final int tamanho = objetos.length;
```

```
3     return new Enumeração {
4        private int posição = 0;
5        public boolean aindaHá( ) {return (posição < tamanho); }
6        public Object próximo( ) throws FaltaElemento {
7           if (posição >= tamanho) throw new FaltaElemento();
8           return objetos[posição++];
9        }
10       public void reInicia( ) { posição = 0; }
11       }
12    }
13 }
```

A boa prática de programação recomenda que uma classe anônima não deve ter mais que algumas linhas de código. 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 construtores. 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 superclassse. Se a classe anônima é derivada de uma interface `I`, a superclasse é `Object`, e a classe anônima é a implementação de `I`. Este 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 o construtor da superclasse `Object`.

Classes anônimas compiladas recebem o nome `NomeClasseExterna$#.class`, onde `#` é um inteiro, que representa o número de ordem da classe, conforme encontrada no programa.

7.6 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 cuidar para que os objetos da classe estendida sejam devidamente associados a objetos da classe envolvente da classe a ser estendida.

A classe estendida pode ou não ser classe interna, em um ou mais níveis, da classe envolvente. No primeiro caso, o objeto envolvente da classe estendida é o mesmo da classe base:

```
1  class A {
2     class B { }
3  }
4  class C extends A {
5     class D extends B {  }
6  }
7  public class Main {
8     public static void main(String[ ] args) {
9        A a = new A();
10       A.B b1 = a.new B();
11       C c = new C();
```

```
12        C.B b2= c.new B( );
13        C.D  d = c.new D();
14     }
15  }
```

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 D.5:

```
1  class A {
2     class B { }
3  }
4  class D extends A.B {
5     D(A a) { a.super(); }
6  }
7  public class Main {
8     public static void main(String[ ] args) {
9        A a   = new A();
10       A.B b = a.new B();
11       C c   = new C();
12       D d   = new D(a);
13    }
14 }
```

7.7 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?

7.8 Notas Bibliográficas

Capítulo 8

Pacotes

Pacote é um recurso para agrupar física e logicamente tipos relacionados, sendo dotado de mecanismo próprio para controle de visibilidade. Pacote corresponde a uma pasta do sistema de arquivo de sistema operacional. Pacote, portanto, é uma coletânea de arquivos, cada um contendo declarações de tipos, i.e., classes e interfaces, 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, obrigatoriamente não-públicas, declaradas em um 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 — classe ou interface —, o nome deste 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. A seguir, 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. O pacote `java.lang`, por ser de uso frequente, dispensa importação explícita; seus tipos são automaticamente importados para toda unidade de compilação.

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` nas declarações de tipos do arquivo `ProgramaGráfico.java`, o qual deve ter a seguinte estrutura:

```
import figuras.*;
public class ProgramaGráfico {
   ...
   usa os tipos importados: Forma, Rectângulo e Círculo
   ...
}
```

onde `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.

8.1 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 arquivo até a pasta que lhe corresponde.

Uma forma de 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 `C:\MeusProgramas\Java\Ed`[1]

Entretanto, a especificação de caminhos absolutos, como a mostrada acima, torna a aplicação muito inflexível e de difícil transporte para outros computadores, mesmo quando operam sob o mesmo sistema operacional. Java resolve este 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 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 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 nesta ordem.

`CLASSPATH` dá o prefixo dos caminhos possíveis e o comando `import` permite adicionar trechos no fim destes prefixos de caminhos para formar o caminho completo que

[1] O caractere "\" representa o símbolo delimitador de pasta usado na especificação de caminhos no sistema de arquivo. Há sistemas operacionais que usam "/"no lugar de "\".

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`, arquivo `t.class`, externo ao pacote ser: `.\A`, `Z:\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 pacote que a contém, conforme definido pela instrução `import`. Exemplo: x.y.z.t
2. substitui os pontos (`"."`) por separadores de diretórios (`"/"` ou `"\"`), conforme a plataforma. Exemplo: `x\y\z\t`
3. acrescenta o sufixo `.class` ao caminho obtido. Exemplo: `x\y\z\t.class`
4. prefixa o resultado com os valores definidos no `CLASSPATH`. Por exemplo: `.\x\y\z\t.class`, `C:\x\y\z\t.class` e `C:\x\y\z\t.class`

O compilador usa os caminhos obtidos acima para 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. Esta indicação pode ser específica, tipo por tipo, ou genérica. 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.

8.2 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` desejado.

A 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:
```
import X.*;
class A { void f( ){ } }
class B extends A {
   A a = new A( );
   a.g( );                    // OK
   a.f( );                    // Erro
   class A{ void g( ){ } }
}
```
2. Tipo `t` é o tipo que está sendo compilado, incluindo tipos herdados. Por exemplo:
```
import X.*;
class A {  ... }
class B extends A {  ... }
class C extends B { ... A ... B ... C ... }
```

As ocorrências de `A`, `B` e `C` durante a compilação de classe `C` acima são dos tipos declarados acima, 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. Nestes casos, deve-se resolver os conflitos usando-se qualificação completa em toda referência aos tipos conflituosos.

8.3 Visibilidade

Pacote é essencialmente um mecanismo de delimitação de visibilidade de tipos e membros de tipos. Tipos não-aninhados que são declarados dentro de pacotes podem ser de acesso público ou acesso pacote. Tipos não-aninhados não podem ser declarados `protected` ou `private`. Acesso público é especificado pelo modificador de acesso `public` e significa que o tipo é também acessível fora do pacote. Acesso pacote, isto é, não-público, é restrito aos tipos locais do pacote. Acesso pacote é obtido pela ausência de modificador de acesso ao tipo.

Membros de classe podem ser declarados `private`, `public`, `protected` ou *pacote*, i.e., 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.

O exemplo a seguir mostra a visibilidade de diversos atributos e métodos das classes de um pacote, onde alguns tem visibilidade pública e outros a têm confinada dentro das fronteiras do pacote onde foram declarados:

- Arquivo `A.java` dentro da pasta `BibliotecaA`:

```
1  package BibliotecaA;
2  public class A {          // visível no pacote e fora
3     private int pri = 1;   // visível somente na classe A
4     public int pub = 2;    // visível onde A for
5     protected int pro = 3; // visível no pacote e em subclasses
6     int pac = 4;           // visível somente no pacote
7  }
8  class B {                 // visível somente no pacote
9     private int pri = 1;   // visível somente na classe B
10    public int pub = 2;    // visível somente no pacote
11    protected int pro = 3; // visível somente no pacote
12    int pac = 4;           // visível somente no 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 }
```

8.4 Compilação com Pacotes

A 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*, argumentos da diretiva `-classpath`, especificam onde encontrar os arquivos `.class` do usuário. A diretiva `-classpath` substitui os caminhos de acesso às classes do usuário especificados pela variável de ambiente `CLASSPATH`. Se nem `-classpath` e nem `CLASSPATH` forem especificados, as classes do usuário são somente as que estiverem no diretório corrente. Se `-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 destes 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 estes 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 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`.

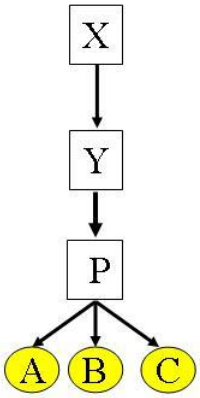


Figura 8.1 Hierarquia de Pacotes I

Considere a hierarquia de diretórios, mostrada na Figura 8.1, cujas folhas são os três arquivos que correspondem a cada uma das linhas abaixo:

```
package P; public class A { ... }  // arquivo A.java
package P; public class B { ... }  // arquivo B.java
package P; public class C {  ... A ... B ... } // arquivo C.java
```

Suponha que as classes `A` e `B` somente usem tipos básicos, e que a classe `C` use somente os tipos `A` e `B`. 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 de `C.java`, a diretiva `-classpath` deve ser usada conforme mostram as linhas de comando abaixo que tratam de disparos de compilação a partir das pastas `P`, `Y` e `X`:

- 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
  ```

  Neste caso, a partir de `X`, pode-se encontrar `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:

```
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.

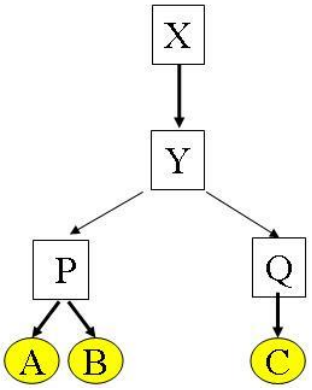


Figura 8.2 Hierarquia de Pacotes II

Para ilustrar a compilação de mais de um pacote, considere a hierarquia de diretórios mostrada na Figura 8.2, contendo os arquivos-fonte:

- 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 disparos de 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 Figura 8.3, onde há duas pastas de nome `P`, que definem uma única biblioteca de nome `P`, contendo as classes `A`, `B` e `M`.

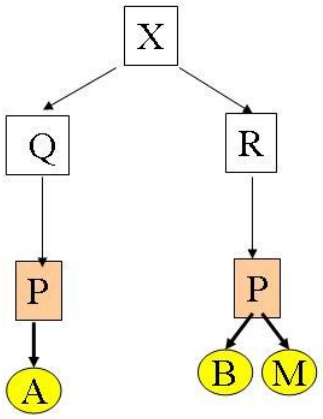


Figura 8.3 Hierarquia de Pacotes III

Suponha que os arquivos-fonte que aparecem na Figura 8.3 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 destes 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.` 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`.

8.5 Exercícios

1. Quais são os mecanismos de Java para controle de visibilidade de um tipo declarado no programa?

8.6 Notas Bibliográficas

Capítulo 9

Entrada e Saída

A bibioteca padrão 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 desta biblioteca pode ser difícil, sendo recomendado seu estudo após se adquirir um pouco mais de fluência na linguagem. É indispensável que operações de entrada e saída possam ser incorporadas nos primeiros programas o mais cedo possível para tornar os exemplos mais atraentes.

Assim, para simplificar o aprendizado, apenas um subconjunto essencial das classes que implementam entrada e saída de fluxo de bytes é apresentado nesta seção, e dessas classes, somente as operações mais simples, mas que sejam suficientes para desenvolver aplicações interessantes.

9.1 Fluxos de *Bytes* e Caracteres

As principais classes básicas que implementam arquivos de fluxo de bytes e caracteres são: `InputStream`, `PrintStream`, `System`, `FilterOutputStream`, `FileInputStream`, `DataInputStream`, `Reader` e `BufferedReader`.

Classe `InputStream`

Essa classe é abstrata e possui operações definidas para ler arquivo de fluxo *byte* a *byte*, retornando o valor inteiro lido no intervalo $[0, 255]$ ou `-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*.

Classe `OutputStream`

Essa é uma classe abstrata com operações para escrever o valor de um `byte`, portanto um valor no intervalo $[0, 255]$, no arquivo de saída de fluxo de *byte*. Ela é usada como superclasse de todas as classe que representam um fluxo de dados de *byte*.

Classe `FilterOutputStream`

Essa classe é uma extensão concreta de `OutputStream`. Sua construtora tem como parâmetro um objeto do tipo `OutputStream`. Objetos de `FilterOutputStream` agem como filtro para objetos `OutputStream`. Suas operações são redefinições das operações de `OutputStream`, que 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 tem `FilterOutputStream` como superclasse.

Class `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.

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 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, e seu valor 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.

Interface `DataInput`

A interface `DataInput` define as assinaturas das várias operações, que devem ser implementadas segundo o contrato definido para cada uma. As principais operações e 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`.

Classe `FileInputStream`

A classe `FileInputStream` é uma subclasse de `InputStream` e permite 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`.

Classe `DataInputStream`

A classe `DataInputStream` é uma subclasse de `FilterInputStream` que implementa a interface `DataInput`. Esta classe permite à aplicação ler valores dos tipos de dados primitivos de Java de um arquivo de fluxo de *bytes*, definido no momento da criação do objeto `DataInputString`. Usualmente uma aplicação grava dados via objetos do tipo `DataOutputStream` e os lê de volta por meio de um objeto `DataInputStream`. As principais operações da classe `DataInputStream` 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`.

O protocolo de uso de `DataInputStream` é ilustrado pelo trecho de 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 se criar, na linha 1, o objeto de acesso ao arquivo, no caso o de nome `arquivo.dat`, e utilizá-lo na construtora de `g`.

Para mostrar o funcionamento de `DataInputStream` e do método `String.split`, muito usados para manipular dados lidos de arquivo de fluxo, suponha que exista o arquivo de fluxo `meusdados.txt` contendo a sequência:

```
ab1c cd2ef,ghi jk3mnop,qrs4tuvz !@#$5%^&*() ZZZ\naaaaa
```

onde `\n` é o caractere de mudança de linha.

O programa `Io` a seguir faz a leitura de uma linha do arquivo de fluxo `meusdados.txt`, a particiona nas maiores subsequências que não contém vírgula (`","`) e imprime estas subsequências separadas por `"--"`:

```
1 public class Io {
2   public static void main (String[ ] args) throws Exception {
3     DataInputStream in;
4     in = new DataInputStream(new FileInputStream("meusdados.txt"));
5     String s = in.readLine();
6     String[] a = s.split("[,]"); // separados por ,
7     for (int x : a)System.out.print(x + "--");
8   }
9 }
```

Saída: `ab1c cd2ef--ghi jk3mnop--qrs4tuvz !@#$%^&*() ZZZ--`

`Reader`

`Reader` é uma subclasse abstrata para leitura de sequências de caracteres em arquivo de fluxo de *bytes*, na forma definida por suas subclasses.

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 UNICODE X *bytes default*.

Recomenda-se o uso desta classe como suporte à entrada de dados realizada por meio da classe `BufferedReader`, segundo o seguinte protocolo:

```
BufferedReader in
       = new BufferedReader(new InputStreamReader(System.in));
```

no qual o teclado é tratado como uma entrada buferizada.

`BufferedReader`

Objetos `BufferedReader` lê 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, arranjos ou de linhas são emitidos. Os comandos de leitura causam pedidos de leitura correspondentes ao fluxo de dados subjacente. Normalmente `BufferedReader` opera sobre fluxos da família `Reader` mais básicos como `InputStreamReader`. As 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 os armazena a partir da posição `p` do arranjo `c`.
- `String readLine()`: lê uma sequência de caracteres até encontrar o fim de linha.

O programa abaixo obtém informações do operador, via o arquivo padrão de entrada `System.in`. O programa espera que o operador digite um número inteiro seguido do pressionamento da tecla *ENTER*. O valor digitado é lido como uma sequência de caracteres, que é convertida para o valor inteiro que representa e é processado adequadamente:

```
1  public class TestaConsole {
2     public static void main(String[] args) throws IOException {
3        Carteira carteira = new Carteira();
4        BufferedReader console = new BufferedReader(
5                        new InputStreamReader(System.in));
6        System.out.println("Quantas moedas de 5 centavos?");
7        String s = console.readLine();
8        carteira.inclua5(Integer.parseInt(s));
9        System.out.println("Quantas moedas de 10 centavos?");
10       s = console.readLine();
11       carteira.inclua10(Integer.parseInt(s));
12       System.out.println("Quantas moedas de 25 centavos?");
13       s = console.readLine();
14       carteira.inclua25(Integer.parseInt(s));
15       double total = carteira.getTotal( );
16       System.out.println("O total é " + total);
17       System.exit(0);
18    }
19 }
20 class Carteira {
21    ...
22    public void inclua5(int n) { ... }
23    public void inclua10(int n) { ... }
```

```
24     public void inclua25(int n) { ... }
25     public int  getTotal( ) { ... }
26  }
```

9.2 Implementação de `myio`

A implementação completa da biblioteca `myio` (vide Seção 1.10), apresentada a seguir, ilustra o uso das classes básicas de entrada e saída de arquivos de fluxos de *bytes* e de caracteres.

Classe `Kbd.java`

```
1  package myio;
2  import java.io.*; import java.util.*;
3  public class Kbd {
4     private static DataInputStream infile =
5                    new DataInputStream(System.in);
6     private static StringTokenizer st;
7     final static public boolean readBoolean() throws IOException {
8        if (st == null || !st.hasMoreTokens())
9            st = new StringTokenizer(infile.readLine());
10       return new Boolean(st.nextToken()).booleanValue();
11    }
12    final static public int readInt() throws IOException {
13       if (st == null || !st.hasMoreTokens())
14           st = new StringTokenizer(infile.readLine());
15       return Integer.parseInt(st.nextToken());
16    }
17    final static public long readLong() throws IOException {
18    if (st == null || !st.hasMoreTokens())
19        st = new StringTokenizer(infile.readLine());
20    return Long.parseLong(st.nextToken());
21    }
22    final static public float readFloat() throws IOException {
23       if (st == null || !st.hasMoreTokens())
24           st = new StringTokenizer(infile.readLine());
25       return new Float(st.nextToken()).floatValue();
26    }
27    final static public double readDouble() throws IOException {
28       if (st == null || !st.hasMoreTokens())
29           st = new StringTokenizer(infile.readLine());
30       return new Double(st.nextToken()).doubleValue();
31    }
32    final static public String readString() throws IOException {
33       if (st == null || !st.hasMoreTokens())
34           st = new StringTokenizer(infile.readLine());
35       return st.nextToken();
```

```
36      }
37  }
```

Classe `Screen.java`

```
1  package myio;
2  public class Screen {
3     final static public void println(){System.out.println();}
4     final static public void print(String s){System.out.print(s);}
5     final static public void println(String s){System.out.println(s);}
6     final static public void print(char c) {System.out.print(c);}
7     final static public void print(int i) {System.out.print(i); }
8     final static public void print(boolean b){System.out.print(b);}
9     final static public void println(boolean b){System.out.println(b);}
10    final static public void print(byte b){System.out.print(b);}
11    final static public void println(byte b){System.out.println(b);}
12    final static public void print(char c){System.out.print(c);}
13    final static public void println(char c){System.out.println(c);}
14    final static public void print(int i){System.out.print(i);}
15    final static public void println(int i){System.out.println(i);}
16    final static public void print(long n){System.out.print(n);}
17    final static public void println(long n){System.out.println(n);}
18    final static public void print(float f){System.out.print(f);}
19    final static public void println(float f){System.out.println(f);}
20    final static public void print(double d){System.out.print(d);}
21    final static public void println(double d){System.out.println(d);}
22 }
```

Classe `InText.java`

```
1  package myio;
2  import java.io.*; import java.util.*;
3  public class InText {
4     private DataInputStream infile;
5     private StringTokenizer st = null;
6     public InText(String fileName) throws IOException {
7        InputStream fin = new FileInputStream(fileName);
8        infile = new DataInputStream(fin);
9     }
10    final public int readInt() throws IOException {
11       if (st == null || !st.hasMoreTokens())
12           st = new StringTokenizer(infile.readLine());
13       return Integer.parseInt(st.nextToken());
14    }
15    final public boolean readBoolean() throws IOException {
16       if (st == null || !st.hasMoreTokens())
17           st = new StringTokenizer(infile.readLine());
18       return new Boolean(st.nextToken()).booleanValue();
19    }
```

```
20     final public byte readByte() throws IOException {
21        if (st == null || !st.hasMoreTokens())
22            st = new StringTokenizer(infile.readLine());
23        return Byte.parseByte(st.nextToken());
24     }
25     final public long readLong() throws IOException {
26        if (st == null || !st.hasMoreTokens())
27            st = new StringTokenizer(infile.readLine());
28        return Long.parseLong(st.nextToken());
29     }
30     final public float readFloat() throws IOException {
31        if (st == null || !st.hasMoreTokens())
32            st = new StringTokenizer(infile.readLine());
33        return new Float(st.nextToken()).floatValue();
34     }
35     final public double readDouble() throws IOException {
36        if (st == null || !st.hasMoreTokens())
37            st = new StringTokenizer(infile.readLine());
38        return new Double(st.nextToken()).doubleValue();
39     }
40     final public String readString() throws IOException {
41        if (st == null || !st.hasMoreTokens())
42            st = new StringTokenizer(infile.readLine());
43        return st.nextToken();
44     }
45  }
```

Classe `OutText.java`

```
1  package myio;
2  import java.io.*;
3  public class OutText {
4    private PrintStream o;
5    public OutText(String fileName) throws IOException {
6       OutputStream fout = new FileOutputStream(fileName);
7       o = new PrintStream(fout);
8    }
9    final public void println() {o.println(); }
10   final public void print(String s) {o.print(s); }
11   final public void println(String s) {o.println(s);}
12   final public void print(byte i) {o.print(String.valueOf(i));}
13   final public void println(byte i) {o.println(String.valueOf(i));}
14   final public void print(int i) {o.print(String.valueOf(i));}
15   final public void println(int i) {o.println(String.valueOf(i)); }
16   final public void print(boolean b) {o.print(String.valueOf(b)); }
17   final public void println(boolean b) {o.println(b + ""); }
18   final public void print(short i) {o.print(String.valueOf(i));}
19   final public void println(short i) {o.println(String.valueOf(i));}
```

```
20    final public void print(long n) {o.print(String.valueOf(n)); }
21    final public void println(long n) {o.println(String.valueOf(n)); }
22    final public void print(float f) {o.print(String.valueOf(f)); }
23    final public void println(float f){o.println(String.valueOf(f));}
24    final public void print(double d) {o.print(String.valueOf(d)); }
25    final public void println(double d){o.println(String.valueOf(d));}
26  }
```

9.3 Exercícios

1. Re-implemente 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.

9.4 Notas Bibliográficas

O texto mais importante para estudar entrada e saída de fluxo é a página de Java mantida pela Sun (http://www.javasoft.com). Além da especificação precisa de cada uma das classes de arquivos de fluxo, os tutorais são altamente esclarecedores.

Capítulo 10

Genericidade

Na linguagem Java, todo objeto é ua instância da classe `Object`, que é a raiz de todas hierarquias de classes. Assim, onde um objeto do tipo `Object` for esperado, pode-se usar objeto 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 encapsulam certos tipos de objetos, mas suas operações não dependem de seus tipos. As operações dessas classes funcionam de maneira uniforme independentemente dos tipos dos objetos por elas encapsulada. Há também o caso de funções que produzem seus resultados independentemente do tipos de seus argumentos. Por exemplo, uma função de ordenação devolve um vetor com seus valores ordenados, não importando o tipo dos seus elementos. Exige-se apenas que eles sejam comparáveis entre si.

Nesses casos, ganha-se genericidade 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 há necessidade de se ter duas implementações para dar conta dessas duas pilhas. Basta que se defina uma classe pilha de `Object` e passe os parâmetros das operações de empilhar e desempilhar adequadamente. Similarmente a função de ordenação, citada acima, pode trabalhar com vetor de `Object`. O mecanismo de polimorfismo de inclusão garante o funcionamento.

Entretanto, o conceito de referências genéricas oferecido por este tipo de polimorfismo transfere ao programador a responsabilidade de garantir o correto uso dos objetos manipulados. O compilador não tem meios de controlar o uso correto dos tipos. Por exemplo, em uma pilha de `Object`, pode-se empilhar tanto um objeto `String`, quando um de uma classe qualquer `A`. 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 estas dificuldades e dar ao compilador os meios de detecção de possíveis erros de mistura indevida de tipo, Java oferece os **tipos e métodos genéricos**.

10.1 Métodos Genéricos

Suponha que você deseja ter as funções de assinatura `void printArray(Integer[ ] a)`, `void printArray(Double[ ] a)` e `void printArray(Character[ ] a)`, para imprimir os conteúdos de vetores dos tipos indicadas. Uma primeira solução seria:

```
1  public static void printArray(Integer[ ] a) {
2     for (Integer e: a) System.out.printf(?%s ?,e);
3     System.out.println();
4  }
5  public static void printArray(Double[ ] a) {
6     for (Double e: a) System.out.printf(?%s ?,e);
7     System.out.println();
8  }
9  public static void printArray(Character[ ] a) {
10    for (Character e: a) System.out.printf(?%s ?,e);
11    System.out.println();
12 }
```

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     printArray(x); printArray(y); printArray(z);
6  }
```

O recurso de métodos genéricos de Java permite trocar as três definições de `printArray` acima por um único método:

```
1  public static <E> void printArray( E[ ] a){
2     for (E e: a) System.out.printf(?%s ?,e);
3     System.out.println();
4  }
```

10.2 Classes Genéricas

Uma classe genérica é uma classe com um ou mais parâmetros formais de tipo, que são designados por identificadores listados entre os pares de colchetes `<` e `>`, colocados após o nome da classe, como `T`, `U`, `S` em `class A<T,U,S>`.

Os tipos parâmetros formais podem, no momento da instanciação da classe genérica, ser substituídos por qualquer tipo da hierarquia `Object`. Dentro da classe genérica, o tipo parâmetro pode ser usado para declarar campos da classe, variáveis locais a funções, parâmetros de métodos, tipos de retorno de métodos e parâmetros de instâncias de classes genéricas.

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 tipo parâmetro.

Objetos do tipo parâmetro podem ser criados, embora, para este fim, exija-se uma pequena variação na sintaxe do operador `new`. Por restrições do compilador Java, um tipo parâmetro de classe genérica não pode ser operando de `new`: 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.

A declaração abaixo mostra a definição da classe `A` de `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 }
```

Observe que o parâmetro formal `T` é utilizado dentro da classe `A` para declarar o campo `x`, o parâmetro `y` do método `s` e o tipo do valor de retorno de `g`.

Os tipos que substituem o parâmetro de tipo da classe genérica no momento da instânciação da classe não podem ser tipos básicos. Caso deseje-se instanciar uma classe genérica com tipos básicos, as correspondentes classes invólucro (*wrapped*) , como `Integer`, associada ao tipo `int`, devem ser usadas no lugar dos tipos primitivos.

A expressão `A<Integer>` cria uma instância da classe `A`, na qual todas as ocorrências do tipo `T` são consistentemente interpretadas como `Integer`. Especificamente, o método `s` da classe `A<Integer>` espera um argumento do tipo `Integer`, e o método irmão `g` retorna um objeto do tipo `Integer`.

O programa abaixo imprime `Valor de i = 100`:

```
1 public class Genérico1 {
2    public static void main(String[ ] args) {
3       A<Integer>m = new A<Integer>( );
4       int k, i = 100;
5       m.s(i);
6       k = m.g();
7       System.out.print("i = " +  k);
8    }
9 }
```

O comando `m.s(i)` somente é válido porque valores de tipos primitivos, e.g., `int`, são automaticamente *encaixotados* em um objeto da classe *wrapped* correspondente, no caso, o tipo `Integer`, sempre que o contexto assim o exigir. O que é passado a `s` é a referência ao objeto do tipo `Integer` criado.

Situação análoga ocorre com a atribuição `k = m.g()`, onde o valor referência a objetos do tipo `Integer` retornado por `m.g()` é automaticamente *desencaixotado* para obter o valor `int` a ser atribuído ao inteiro `k`.

No programa `Genérico2` abaixo, que imprime `Valor de i = 10, Valor de t = 100.0`, a classe genérica `A` é instanciada duas vezes, com os tipos `Integer` e `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);
```

```
6         k = m.g();
7         System.out.print("Valor de i = " + k + ", ");
8         A<Double> q = new A<Double>( ) ;
9         double w, t = 100.0;
10        q.s(t);
11        w = q.g();
12        System.out.print("t = " + w);
13     }
14  }
```

Note que, embora objetos do tipo `A<Integer>` ou `A<Double>` possam substituir objetos do tipo `A<T>`, os tipos `A<Integer>` e `A<Double>` são distintos e incompatíveis, não tendo qualquer relação entre si, exceto que ambos sejam instâncias de um mesmo tipo base e portanto subtipos deste tipo base.

É possível utilizar uma classe genérica para declarar objetos sem especificar seus argumentos de tipo. Neste caso, é criado uma instância da classe genérica usando implicitamente o tipo `Object` com argumento. Os objetos assim declarados têm **tipos brutos**. Por exemplo, `A a = new A( )` é equivalente a `A<Object> a = new A<Object>( )`.

A uma referência a objetos de tipo bruto pode-se atribuir referências a objetos de classe genérica com argumentos, como é feito em `A a = new A<Integer>( )`. A atribuição invertida, `A<Integer> b = a`, é permitida, mas deve ser evitada, pois é perigosa, uma vez que não se pode garantir que os objetos apontados por `a` armazenam sempre objetos `Integer`.

O programa `Genérico3` tem erro de tipo e se submetido ao compilador Java receberá a mensagem "`incompatible type em q.s(t) e q.g()`".

```
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 }
```

10.3 Hierarquia de Genéricos

Classes genéricas podem ser estendidas para formar novas classes, genéricas ou não. A declaração

```
class B extends A<Double> { ... }
```

cria uma nova classe de nome `B` a partir da instância `A<Double>` de `A<T>`. Referências a objetos do tipo `B` podem ser usadas quando esperam-se referências a objetos do tipo `A<T>` ou `A<Double>`.

O programa `Genérico4` produz o mesmo resultado que `G2`:

```
1  public class Genérico4 {
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("i = " + k + ", ");
7        B q = new B( );
8        double w, t = 100.0;
9        q.s(t);
10       w = q.g( );
11       System.out.print("t = " + w);
12    }
13 }
```

No momento da instanciação de uma classe genérica, os parâmetros fornecidos devem ser tipos válidos no escopo. O mesmo ocorre quando uma classe genérica é estendida ou especializada. Exemplos de especialização correta são:

```
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 tipos parâmetros da classe genérica derivada, e.g., `H` de `C` e `T` e `R` de `D`, podem ser usados para definir a instância da classe base desejada.

Por outro lado, a declaração

```
class F extends A<T> {  }
```

está sintaticamente errada pela indefinição do tipo `T`, usado para instanciar `A`.

O programa `Genérico5`, que ilustra o uso das classes genéricas `A` e `D`, imprime

`i = 10, b = 100, c = 1000.0`.

```
1  public class Genérico5 {
2     public static void main(String[ ] args) {
3        A<Integer> m = new A<Integer>( );
4        int k,i = 10;
5        m.s(i);
6        k = m.g();
7        System.out.print("i = " + k + ", ");
8        double c = 1000.0;
9        D<Integer,Double> q = new D<Integer,Double>(c);
10       int a, b = 100;
11       double t;
12       q.s(b);
13       a = q.g( );
14       System.out.print("b = " + a + ", ");
15       t = q.h( );
```

```
16        System.out.print("c = " + t);
17    }
18 }
```

Por garantir a compilação de programas de versões Java mais antigas pelo 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> { }
```

Esta convenção permite que programas escritos em versões Java anteriores a 5.0 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`, como `LinkedList`, foram substituídas por versões parametrizadas do tipo `LinkedList<T>`.

10.4 Interfaces Genéricas

Interfaces genéricas são interfaces com um ou mais parâmetros do tipo **tipo**, 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 para a definição de classes não-genéricas ou genéricas. Um exemplo do uso de interface para implementar classes é ilustrado pelas seguintes implementações das classes 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 `Genérico6` abaixo imprime `t = 10`:

```
1 public class Genérico6 {
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 }
```

10.5 Limite Superior de Tipo

Parâmetros de classes genéricas podem corresponder a objetos pertencente a qualquer nível da hierarquia das referências a `Object`. Assim, somente as operações aplicáveis a `Objecct` são garantidamente válidas em todo uso da classe genérica.

Entretanto há situações que se deseja restringir os parâmetros a hierarquias específicas, permitindo que outras operações, além daquelas definidas por `Object`, possam ser aplicadas a referências do tipo parâmetro da classe genéricas. O recurso de Java para este fim denomina-se **Parâmetro com Limite Superior**, que permite a especialização do tipo parâmetro.

No exemplo abaixo, as operações aplicáveis a objetos do tipo `T` são as definidas para a classe `R`, no caso, o método `f`:

```
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  public class R {
7     public int f( ) {  return 1; }
8  }
9  public class R1 extends R{
10    public int g( ){return 100;  }
11 }
12 public class R2 extends R{
13    public int f( ){return 1000; }
14 }
```

No exemplo acima, `R` é uma classe, mas poder-se-ia usar em seu lugar uma interface, conforme 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  interface R { public int f( ) }
7  class R1 implements R {
8     public int f( ){return 1; }
9     public int g( ){return 100; }
10 }
11 class R2 implements R {
12    public int f( ){return 1000; }
13 }
```

O programa `Genérico7`, onde `M` pode ser qualquer uma das implementações acima, imprime `k = 100, j = 1000`:

```
1  public class Genérico7 {
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);
9         r.s(b);
10        k = m.g();
11        j = r.g();
12        System.out.print("k = "+ k + ", j = "+ j);
13     }
14  }
```

Tentativas de se 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 Genérico8{
2     public static void main(String[ ] args) {
3        M<Integer> m = new M<Integer>( );   // ERRO DE COMPILAÇÃO
4        int j,k = 1;
5        m.s(k);
6        j = r.g( );
7        System.out.print("k = "+ k + ", j = " +  j);
8     }
9  }
```

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 ter 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);}
```

Contudo, essa interface já faz parte da biblioteca de Java, não sendo necessário sua implementação.

O método `max` pode então ser definido da forma abaixo:

```
1  class Comparadores {
2     public static <T extends Comparable<T> > T max( T x, T y, T z) {
3        T m = x;
4        if (y.compareTo(m) > 0)  m = y;
5        if (z.compareTo(m) > 0)  m = z;
6        return m;
7     }
8     public static void main(String[ ] args) { ... }
```

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` abaixo, que ilustra usos de `max`, imprime os valores `5`, `5.0` e `W. Mozart`.

```
1 public static void main(String[ ] args) {
2    System.out.print(max(3, 4, 5));
3    System.out.print(max(3.3, 5.0, 4.0));
4    System.out.print(max("W. Mozart", "L. Beethoven", "A. Vivaldi"));
5 }
```

10.6 Tipo Curinga

10.7 Exercícios

1. Mostre como objetos contêiner podem sem implementados em Java sem o uso de classes.

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.

10.8 Notas Bibliográficas

Deitel [8] apresenta uma introdução a classes genéricas de Java por meio de muitos e bons exemplos de programação.

Bertrand Meyer [33] discute vantagens e desvantagens do uso de genericidade e herança como mecanismo para favorecer reusabilidade.

Capítulo 11

Coleções

11.1 Introdução

11.2 Exercícios

11.3 Notas Bibliográficas

Capítulo 12

Interface Gráfica do Usuário

GLOSSÁRIO

GUI - *Grafical User Interface*: é a interface gráfica de um programa com a qual o usuário pode interagir.

Componente : um objeto gráfico com o qual o usuário interage via mouse ou teclado.

AWT - *Abstract Windowing Toolkit*: um pacote de classes Java que disponibiliza ao programador uma série de classes relacionadas com as capacidades de *GUI* da plataforma local.

Swing : um pacote gráfico Java, mais moderno que o AWT, cujos componentes são escritos, manipulados e exibidos completamente em Java. Em geral o comportamento de componentes AWT depende da plataforma, ao passo que o comportamento dos componentes Swing não depende dela.

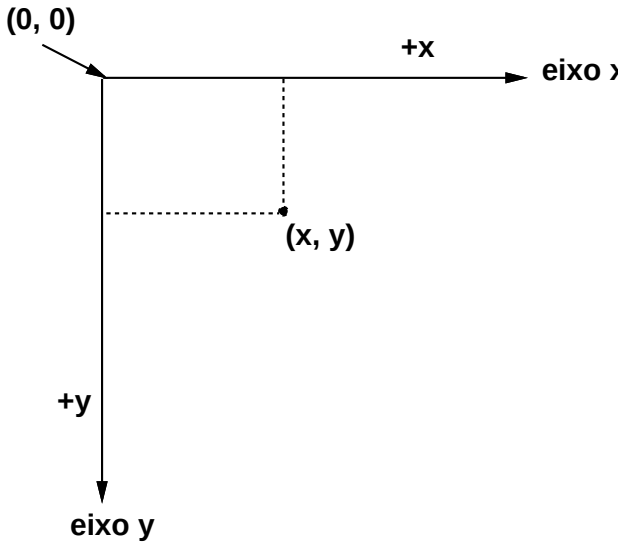


Figura 12.1 Sistema de Coordenadas da Tela

- As unidades são medidas em *Pixels*.
- *Pixel* - A menor resolução do monitor.
- (0, 0) - Canto superior esquerdo de uma janela.

12.1 Janelas

- A classe **JFrame** permite a construção de janelas.
- A classe **Graphics** é usada para desenhar formas sobre componentes gráficos.
- A classe `Polygon` é usada para criar polígonos
- A classe **Font** controla a aparência das fontes disponíveis.
- A classe **Color** define a cor dos elementos gráficos.
- Uma *Frame* é uma janela que possui decorações como bordas, barra de título e botões para fechar e iconificar.
- Em Java, frames podem ser objetos da classe **JFrame**.
- Aplicativos que possuem uma interface gráfica usam frames para definir janelas.
- Classes gráficas baseadas no pacote **javax.swing** geralmente herdam as propriedades da classe **JFrame**.
- O estilo de uma janela, que é um objeto do tipo **JFrame**, depende da plataforma (Ex.: Windows, UNIX, etc).
- Pode-se desenhar sobre um objeto **JFrame** e adicionar-lhe componentes gráficos, como botões, rótulos, caixas de texto.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
           |      |
           |      +--java.awt.Window
           |               |
           |               +--java.awt.Frame
           |                       |
           |                       +--javax.swing.JFrame
           +--javax.swing.JComponent
                |
                +--- javax.swing.JPanel
```

MÉTODOS DA CLASSE JFRAME I

- `setTitle(String t)`:
  define título da janela
- `setSize(int largura, int altura)` ou `setSize(Dimension d)`:
  define dimensões da janela
- `show( )`:
  coloca janela visível e por cima das demais

- `setVisible(boolean b)`:
  controla visibilidade da janela
- `setLocation(int x, int y)` ou `setLocation(Point p)`:
  posiciona a janela na tela

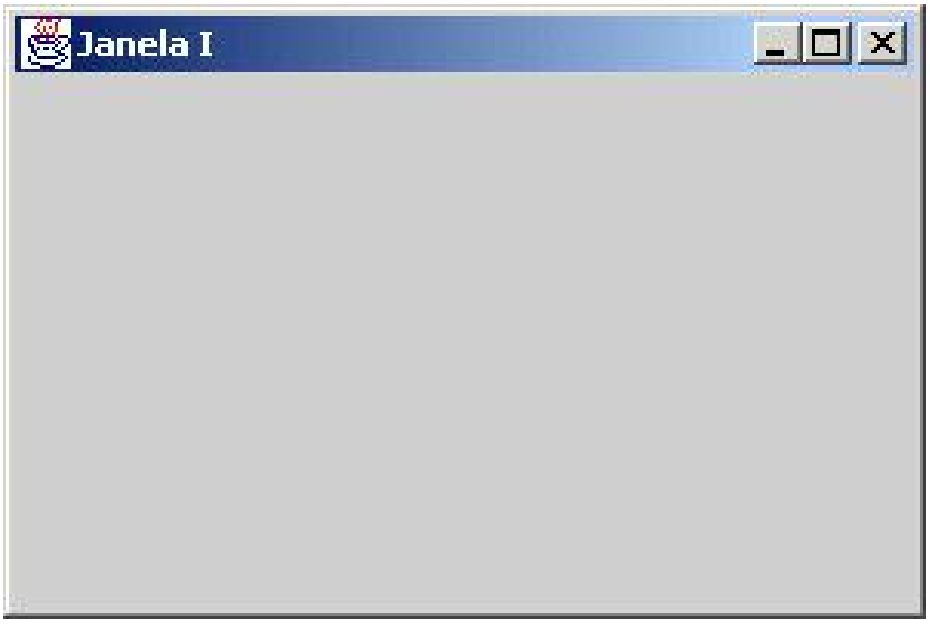
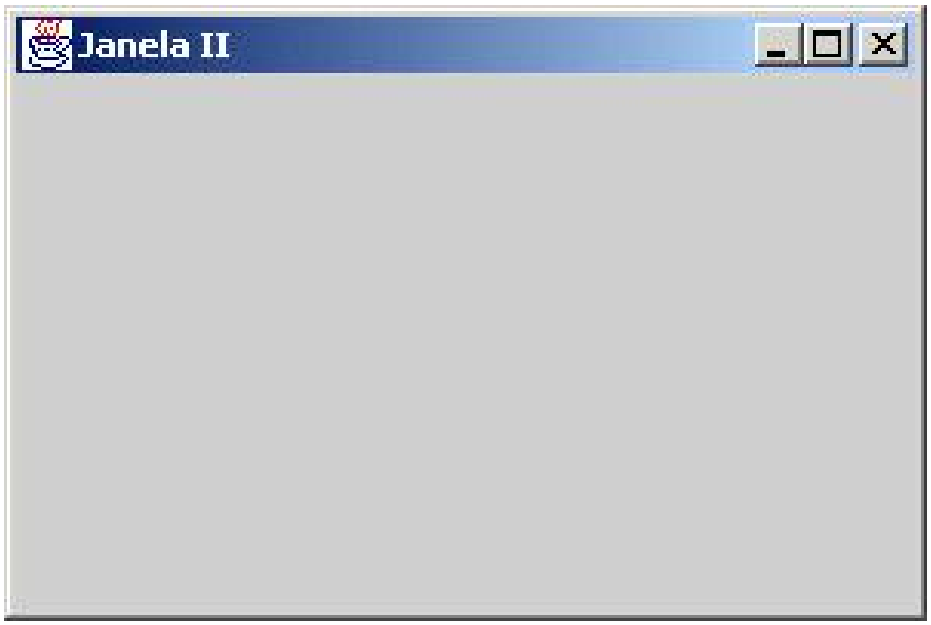


Figura 12.2 Primeira Janela

CRIAÇÃO DA PRIMEIRA JANELA

```
import javax.swing.*;

class Janela extends JFrame {
   public Janela ( ) {
      setTitle("Janela I");
      setSize(300,200);
  }
}
public class UsaPrimeiraJanela {
   public static void main(String [ ] args) {
      JFrame janela = new Janela( );
      janela.setVisible(true);
   }
}
```

CRIAÇÃO DE JANELA QUE FECHA




```
import javax.swing.*; import java.awt.event.*;
class Janela extends JFrame {
   public Janela (){setTitle("Janela II"); setSize(300,200);}
}
class Ouvinte extends WindowAdapter{
   public void windowClosing (WindowEvent e){System.exit(0);}
}
public class UsaSegundaJanela {
   public static void main(String [ ] args) {
      JFrame janela = new Janela( );
      Ouvinte a = new Ouvinte( ); janela.addWindowListener(a);
      janela.setVisible(true);
   }
}
```

OUTRA FORMA DE ADICIONAR O OUVINTE

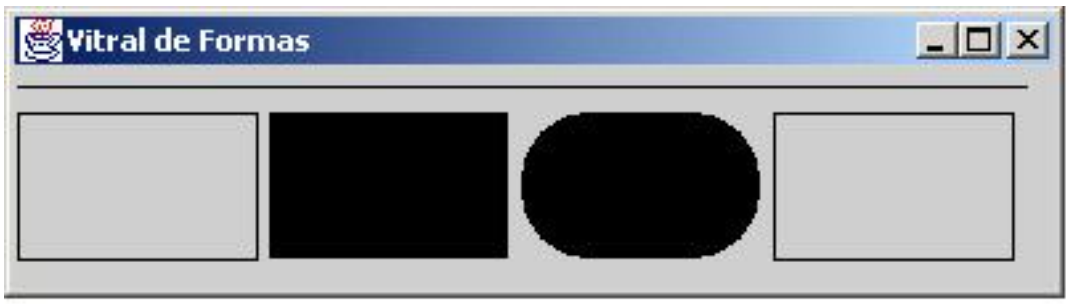
```
import javax.swing.*; import java.awt.event.*;
class Janela extends JFrame {
   public Janela (){setTitle("Janela II"); setSize(300,200);}
}
public class UsaSegundaJanela {
   public static void main(String [ ] args) {
      JFrame janela = new Janela( ); janela.setVisible(true);
      janela.addWindowListener(
         new WindowAdapter{
             public void windowClosing (WindowEvent e)
                 {System.exit(0);}
```

```
        }
    );
} }
```

MÉTODOS DA CLASSE JFRAME II

- `paint(Graphics g)`:
  - Desenha informações gráficas na janela.
  - É chamado automaticamente, por exemplo, em resposta a um *evento*, como *click* de mouse ou abertura de janela.
  - Deve ser redefinido pelo programa do usuário
  - Não deve ser chamado diretamente
  - `paint(Graphics g)` é da classe `Container`
- `repaint()`:
  - Limpa o fundo do componente gráfico corrente de qualquer desenho anterior e chama o método **paint** para redesenhar as informações gráficas sobre tal componente.
  - Método `repaint()` pode ser chamado, quando necessário.

CRIAÇÃO DE OUTRA JANELA

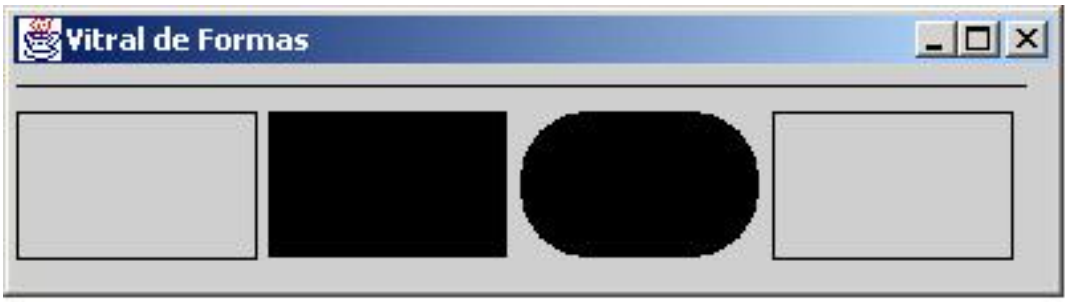



```
import java.awt.*; import javax.swing.*;
public class Vitral extends JFrame {
    public Vitral( ) {
        super ("Vitral de Formas"); setSize (400, 110);
        show();  // setVisible(true);
    }
    public void paint (Graphics g) {
        g.drawLine (5, 30, 380, 30);
        g.drawRect (5, 40, 90, 55);
        g.fillRect (100, 40, 90, 55);
        g.fillRoundRect (195, 40, 90, 55, 50, 50);
```

```
        g.draw3DRect (290, 40, 90, 55, true);
    }
}

import java.awt.*; import javax.swing.*;
import java.awt.event.*;
public class UsaVitral {
    public static void main (String args[]) {
        Vitral j = new Vitral( );
        j.addWindowListener(
            new WindowAdapter () {
                public void windowClosing (WindowEvent e) {
                    System.exit(0);
                }
            }
        );
    }
}
```

SAÍDA DE `Vitral`

```
>java UsaVitral
```



SEQÜÊNCIA DE AÇÕES DE `UsaVitral`

- Construtora de `JFrame` cria a estrutura que representa a janela
- Coloca-se título na janela Vitral
- Define-se o tamanho da janela
- Torna a janela visível, causando chamada ao `paint()`
- Associa-se objeto ouvinte ao objeto Vitral

MÉTODOS DA CLASSE JFRAME III

- `dispose ( )`:
  fecha janela e recupera os recursos do sistema utilizados
- `setIconImage(Image imagem)`:
  define a imagem da janela iconizada ou minimizada
- `setBounds(intx, int y, int largura, int altura)`:
  posiciona janela na tela e define suas dimensões

- `Point getLocation( )`:
  devolve coordenadas do vértice esquerdo superior da janela

12.2 Desenhos Geométricos

CLASSE `Graphics`

- A classe `Graphics` é abstrata.
- Cada ambiente implementa uma classe derivada de `Graphics` com todos os recursos de desenho.
- Objetos `Graphics` controlam como os desenhos são feitos.
- Objetos `Graphics` permitem:
  - desenhar formas geométricas;
  - manipular estilos, tamanhos e tipos de fontes;
  - manipular cores;
- Objeto do tipo `Graphics` criado pela GUI e passado automaticamente ao `paint`

Métodos

- `public void drawLine (int x1, int y1, int x2, int y2)`
  - `g.drawLine(x1,x2,y1,y2)` desenha no objeto `g` do tipo `Graphics` uma linha entre os pontos **(x1, y1)** e **(x2, y2)**
- `public void drawRect (int x, int y, int width, int height)`
  - Desenha um retângulo com a largura (**width**) e a altura (**height**) especificadas.
- `public void fillRect(int x, int y, int width, int height)`
  - Desenha um retângulo sólido com a largura e altura especificadas, com canto superior esquerdo no ponto **(x, y)**.
- `public 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.
- `public drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)`
  - Desenha um retângulo com cantos arredondados.
- `public 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.
- `public void draw3DRect (int x, int y, int width, int height, boolean b)`

  - Desenha um retângulo tridimensional em algo relevo quando **b** é **true** e em baixo relevo quando **b** é **false**.

- `public 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`.
- `public void drawOval (int x, int y, int width, int height)`
  - Desenha uma oval 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.
- `public void fillOval (int x, int y, int width, int height)`
  - Desenha uma oval com o fundo preenchido na cor atual.

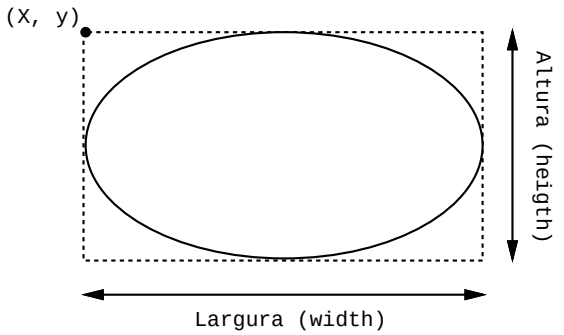
CONTROLE DE CORES

- `public Color getColor( )`
  devolve um objeto Color que representa a cor de textos e linhas gráficos
- `public void setColor (Color c)`
  Configura a cor atual para desenho de textos e linhas gráficos.

DESENHO DE ARCOS

- `public void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)`
  - Desenha um arco em relação às coordenadas do canto superior esquerdo do retângulo delimitador `(x, y)` com a largura (`width`) e a altura (`height`) especificadas.
  - O segmento de arco é desenhado iniciando em `startAngle` e varrendo `arcAngle` graus.

- `public void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)`



– Desenha um arco sólido (isto é, um setor) em relação às coordenadas do canto superior esquerdo do retângulo delimitador **(x, y)**.

DESENHO DE POLÍGONOS

- `public void drawPolyline(int xPoints[], int yPoints[], int npts)`
  - Desenha uma série de linhas conectadas.
  - Se o último ponto definido por um par `(x, y)`, (x ∈ **xValues** e y ∈ **yValues**) for diferente do primeiro ponto, a polilinha não será fechada.
  - `npts` dá o número de pontos

- `public void drawPolygon (Polygon p)`
  - Desenha o poligono fechado especificado .
- `public void fillPolygon (Polygon p)`
  - Desenha o polígono especificado preenchido com a cor corrente do contexto gráfico.
- `public void fillPolygon(int xPoints[],int yPoints[],int npts)`
  - Desenha um polígono sólido cujas coordenadas são especificadas pelos *arrays* **xPoints** e **yPoints**.
  - O último argumento especifica o número de pontos.
  - Esse método desenha um polígono fechado – mesmo se o último ponto for diferente do primeiro ponto.
- `public void drawPolygon(int xPoints[],int yPoints[],int npts)`
  - Desenha um polígono.
  - A abscissa `x` de cada ponto é especificada pelo array `xPoints` e a ordenada `y` de cada ponto é especificada no array `yPoints`.
  - O último argumento especifica o número de pontos, ou seja, o número de vértices do polígono.
  - Este método desenha um polígono fechado, mesmo que o último ponto seja diferente do primeiro ponto.

Controle de Cores

- **public Font getFont( )**
  - Retorna uma referência de objeto **Font** representando a fonte atual.
- **public void setFont(Font f)**
  - Configura a fonte atual com a fonte, o estilo e o tamanho especificados pela referência ao objeto **Font f**.

Classe `Polygon`

MÉTODOS PARA CRIAR POLÍGONOS

- `public Polygon()`:
  - Constrói um novo objeto do tipo polígono. O polígono recém contruído não contém nenhum ponto.
- `public Polygon(int xValues[], int yValues[], int npts)`:
  - Constrói um novo objeto do tipo polígono com `npts` vértices, sendo cada vértice é formado por uma coordenada `x` do *array* `xValues` e uma coordenada `y` do *array* `yValues`.
- `addPoint(int x, int y)`:
  Adiciona um novo ponto ao polígono

```
Polygon p = new Polygon ( );
p.addPoint(10,10); p.addPoint(10,30); p.addPoint(20,30);
g.drawPolygon(p);
```

Classe `Font`

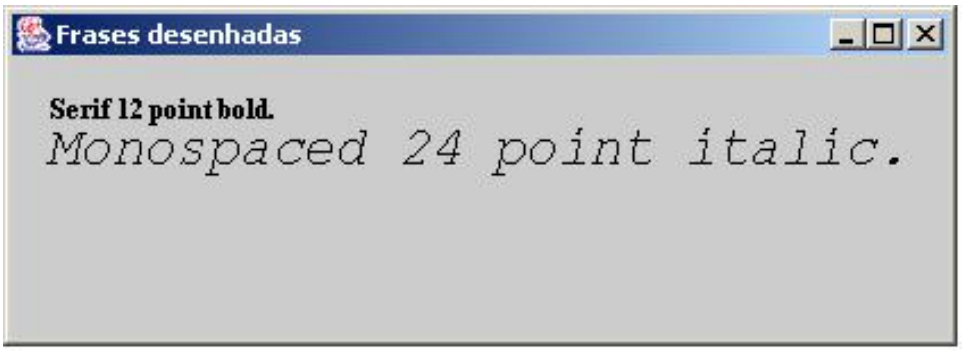
MÉTODOS E CONSTANTES DE `Font`

- `public final static int PLAIN`
  - Uma constante representando um estilo de fonte simples.
- `public static int BOLD`
  - Uma constante representando um estilo de fonte em negrito.
- `public static int ITALIC`
  - Uma constante representando um estilo de fonte em itálico.
- `public Font (String name, int style, int size)`
  - Cria um objeto **Font** com a fonte, o estilo e o tamanho especificados.
  - `name` pode ser um dos nomes lógicos: `"Monospaced"`, `"SansSerif"`, `"Serif"`, `"Dialog"`, `"DialogInput"`
    Fontes acima são mapeadas para as existentes na máquina.
    Por exemplo, no Windows `"SansSerif"` é mapeado para Arial
  - `style` pode ser `Font.PLAIN, Font.ITALIC, Font.BOLD`
  - `style` pode combinações: `Font.ITALIC + Font.BOLD`
  - `size` em pontos (1 ponto = 1/72 da polegada)
- **public int getStyle()**
  - Retorna um valor inteiro indicando o estilo da fonte.
- **public int getSize()**
  - Retorna o tamanho da fonte.
- **public String FontName()**

  - Retorna o nome da fonte, por exemplo `"Helvetica Bold"`

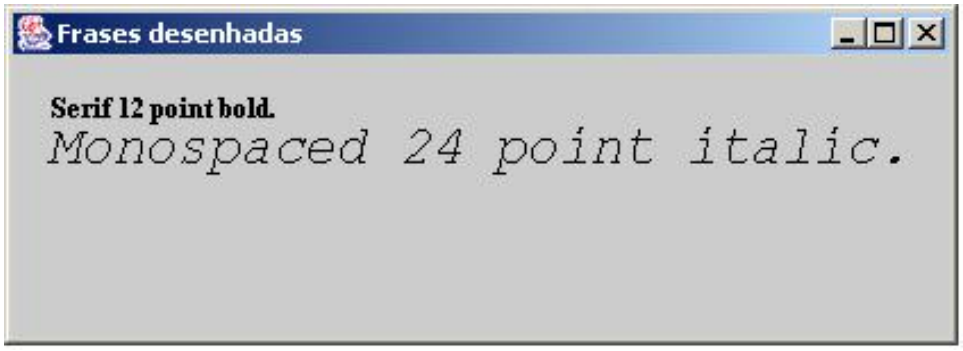
- **public String getName()**
  - Retorna o nome lógico da fonte, por exemplo `"SansSerif"`
- **public String getFamily()**
  - Retorna o nome da família da fonte, por exemplo,`"Helvetica"`
- **public boolean isPlain ()**
  - Testa uma fonte quanto a um estilo de fonte simples. Retorna **true** se o estilo da fonte é simples (PLAIN).
- **public boolean isBold**
  - Testa uma fonte quanto a um estilo de fonte em negrito. Retorna **true** se a fonte estiver em negrito.
- **public boolean isItalic()**
  - Testa uma fonte quanto a um estilo de fonte em itálico. Retorna **true** se a fonte estiver em itálico.

DEMONSTRANDO USO DE `Font`




```
import java.awt.*;
import javax.swing.*;

public class DesenhaFrase extends JFrame {
    public DesenhaFrase() {
        super ("Frases desenhadas");
        setSize (420, 150);
        setVisible(true);
    }
    public void paint (Graphics g) { ...}
}

public void paint (Graphics g) {
    g.setFont (new Font ("Serif", Font.BOLD, 12));
    g.drawString ("Serif 12 point bold.", 20, 50);
```

```
    g.setFont (new Font ("Monospaced", Font.ITALIC, 24));
    g.drawString ("Monospaced 24 point italic.", 20, 70);
}
```




```
import java.awt.*; import javax.swing.*;
import java.awt.event.*;
public class UsaDesenhaFrase {
   public static void main (String args[]) {
      DesenhaFrase j = new DesenhaFrase();
      j.addWindowListener(
          new WindowAdapter () {
              public void windowClosing (WindowEvent e) {
                   System.exit(0);
               }
            }
        );
   }
}
```

12.3 Controle de Cores

Classe `Color`

MÉTODOS DE `Color`

- **public Color (int r, int g, int b)**
  - Cria uma cor com base no conteúdo de vermelho, verde e azul, expressos como inteiros entre 0 e 255.
- **public Color (float r, float g, float b)**
  - Cria uma cor com base no conteúdo de vermelho, verde e azul, expressos como valores de ponto flutuante entre 0.0 e 1.0

CONSTANTES DE COR

Constante Color	Cor	Valor RGB
public final static Color orange	laranja	255,200,0
public final static Color pink	cor-de-rosa	255,175,175
public final static Color cyan	ciano	0,255,255
public final static Color magenta	magenta	255,0,255
public final static Color yellow	amarelo	255,255,0
public final static Color black	preto	0,0,0
public final static Color white	branco	255,255,255
public final static Color gray	cinza	128,128,128
public final static Color lightGray	cinza-claro	192,192,192
public final static Color darkGray	cinza-escuro	64,64,64
public final static Color red	vermelho	255,0,0
public final static Color green	verde	0,255,0
public final static Color blue	azul	0,0,255

OUTROS MÉTODOS DE `Color`

- **public int getRed()**
  - Retorna um valor entre 0 e 255 representando o conteúdo de vermelho
- **public int getGreen()**
  - Retorna um valor entre 0 e 255 representando o conteúdo de verde
- **public int getBlue()**
  - Retorna um valor entre 0 e 255 representando o conteúdo de azul

USO DE `Color`

```
import java.awt.*; import javax.swing.*;
public class RectColorido extends JFrame {
    public RectColorido() {
        super ("Retângulos coloridos");
        setSize (150, 80);
        setVisible(true);
    }
  public void paint (Graphics g) {
       g.setColor (new Color (255, 0, 0)); // vermelho
       g.fillRect (25, 25, 100, 20);
       g.setColor (new Color (0.0f, 1.0f, 0.0f )); // verde
       g.fillRect (25, 50, 100, 20);
  }
}
```

DEMONSTRANDO USO DE `Color`

```
import java.awt.*; import javax.swing.*;
import java.awt.event.*;
public class UsaRectColorido {
    public static void main (String args[]) {
```

```
        RectColorido j = new RectColorido();
        j.addWindowListener(
           new WindowAdapter () {
               public void windowClosing (WindowEvent e) {
                    System.exit(0);
                }
           }
        );
    }
}
```

DEMONSTRANDO USO DE `Color`

```
>java UsaRectColorido
```

12.4 Componentes Básicos de Interface

COMPONENTES BÁSICOS

- Vimos como utilizar objetos da classe `Graphics` para desenhar formas geométricas e textos coloridos.
- Os programas vistos até então não dispunham de muitas formas de interagir com o usuário.
- A seguir serão estudados métodos e subclasses da classe `Component`, que possibilitam ao usuário interagir com programas gráficos em Java.
- Componentes GUI são objetos com os quais o usuário interage via mouse, teclado, voz, etc.

COMPONENTE `JLabel`

- `JLabel` (rótulo):
  - Uma área em que podem ser exibidos textos não-editáveis ou ícones.
  - *Labels* ou *rótulos* são utilizados para mostrar um pequeno texto que não pode ser alterado.
  - Rótulos comuns são vistos nomeando botões e caixas de texto.

- Métodos de `JLabel`:
  - `JLabel( )`: rótulo sem identificação
  - `JLabel(String t)`: cria rótulo com identificação `t`
  - `void setText(String t)`: define a identificação do rótulo
  - `String getText( )`: devolve a identificação do rótulo

COMPONENTE `JButton`

- `JButton` (botão):
  Uma área que aciona um evento quando recebe um *click* de mouse.
- Métodos de `JButton`:
  - `JButton( )`: cria botão sem identificação
  - `JButton(Icon icon)`: cria botão com o ícone
  - `JButton(String text)`: cria botão com o texto

COMPONENTE `JTextField`

- `JTextField` (caixa de texto):
  - Uma área em que o usuário insere dados via teclado. A área também pode exibir informações.
  - *Caixas de Texto* geralmente são uma parte de uma janela onde texto pode ser escrito.
  - Caixas de texto são um recurso utilizado nas interfaces gráficas para receber texto do usuário ou para transmitir texto para ele.
- Métodos de `JTextField`:
  - `JTextField( )`
  - `JTextField(int colunas)`
  - `JTextField(String texto)`

COMPONENTES `JComboBox`, `JList`, `JPanel`

- `JComboBox` (caixa de combinação):
  Uma lista *drop-down* de itens a partir da qual o usuário pode fazer uma seleção clicando em um item na lista ou digitando na caixa, se permitido.
- `JList` (lista):
  Uma área em que uma lista de itens é exibida 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. Múltiplos elementos podem ser selecionados.
- `JPanel` (painel):
  Um contêiner em que os componentes podem ser colocados.

IMPORTANTES DE COMPONENTES SWING

- Para utilizar efetivamente os componentes GUI, as hierarquias de herança dos pacotes **javax.swing** e **java.awt** devem ser compreendidas
- Deve-se conhecer especialmente as classes:
  - **Component**
  - **Container**
  - **JComponent**

  que definem os recursos comuns à maioria dos componentes Swing.

HIERARQUIA DE JComponent

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
           |      |
           |      +--java.awt.Window
           |               |
           |               +--java.awt.Frame
           |                       |
           |                       +--javax.swing.JFrame
           +--javax.swing.JComponent
                |
               ...

  javax.swing.JComponent
  |  | ... |  |   |   |
  |  |     |  |   |   +--- JPanel
  |  |     |  |   +-- JComboBox
  |  |     |  +--- Jlist
  |  |     +--- JLabel
  |  |
  |  +--- AbstractButton
  |           |
  |           +--- JButton, JToggleButton, JMenuItem
  +-- JTextComponent            |
         |                      + JRadioButton, JCheckBox
         +--- JTextField, JTextArea
```

CLASSE `Container`

- Um **container** é uma coleção de componentes relacionados.
- Em aplicativos com **JFrames** e *applets*, anexamos os componentes ao painel de conteúdo – um **Container**.

- **Containers** possuem um leiaute que define como componentes gráficos serão posicionados em seu espaço.

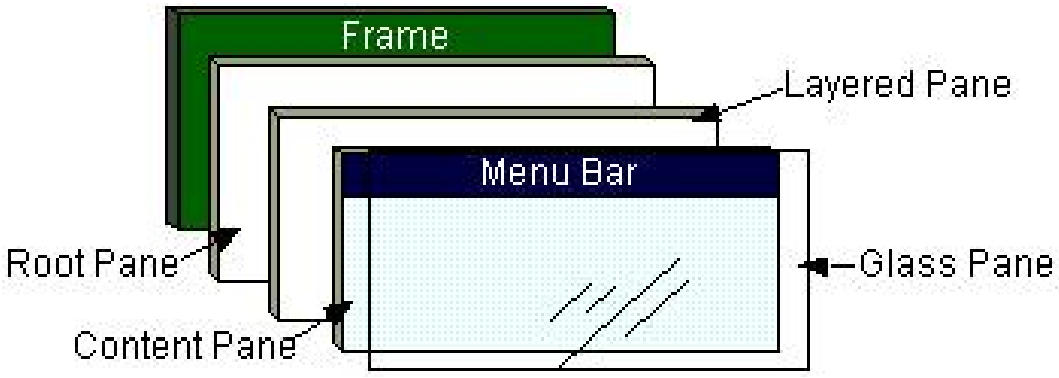
MÉTODOS DE `Container` I

- `add(Component c)`:
  adiciona o componente `c` no fim do contêiner
- `add(Component c, int posicao)`:
  adiciona o componente `c` na posição indicada do contêiner
- `setLayout(LayoutManager leiaute)`:
  define o leiaute a ser usado para posicionamento de componentes no contêiner. Leiaute definido por objeto do tipo:
  - `FlowLayout`
  - `BorderLayout`
  - `GridLayout`
- `setFont(Font f)`:
  define o fonte do contêiner

12.5 Gerência de Leiaute

ESTRUTURA DE UMA JANELA (`JFrame`)

- Uma janela é uma estrutura com várias camadas:



- `ContentPane` é usado para adicionar novos componentes à janela:

```
JFrame janela = new MinhaJanela();
Container c = janela.getContentPane();
c.setLayout (new FlowLayout( ));
c.add(componente);
```

ORGANIZAÇÃO DE COMPONENTES EM UMA GUI

- Posicionamento Absoluto:
  Sob o total controle pelo programador
- Gerente de Layout:
  Posicionamento automático
- Programação Visual com IDE:
  Caixa de ferramenta + arrastar e colar

POSICIONAMENTO ABSOLUTO

- É obtido configurando o leiaute do contêiner como `null`.
- Permite maior nível de controle da aparência do GUI.
- Permite especificar a posição absoluta de cada componente em relação ao canto superior esquerdo.
- É necessário que se especifique o tamanho de cada componente.

```
...
JFrame f = new MinhaFrame();
Component meuComponente = new MeuComponente( );
Container c = f.getContentPane();
c.setLayout(null);
c.add(meuComponente);
meuComponente.setBounds(x,y,largura,altura);
...
```

GERÊNCIA DE LEIAUTES

- *Gerentes de Leiaute* se prestam a organizar componentes GUI em um contêiner para propósito de apresentação.
- Retira-se do programador a tarefa de posicionar os itens em um contêiner fazendo o posicionamento automaticamente.
- Muitos ambientes de programação Java fornecem ferramentas GUI que ajudam o desenvolvedor a controlar a aparência de suas aplicações.
- Existem várias classes para gerenciar automática do leiaute:
  FlowLayout, BorderLayout, `GridLayout`, etc

CLASSE `FlowLayout`

- Dispõe os componentes seqüencialmente (da esquerda para a direita) na ordem que foram adicionados.
- Também é possível especificar a ordem dos componentes utilizando o método **add** de **Container** que aceita um **Component** e uma posição de índice inteiro como argumento.

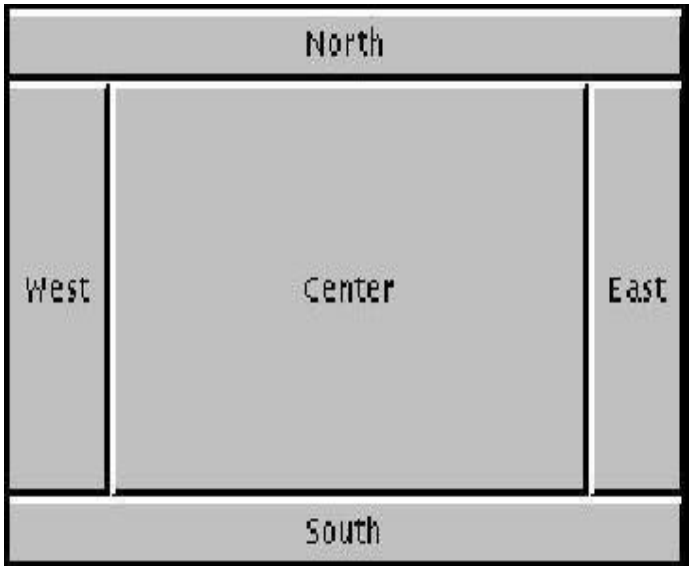
```
...
JFrame f = new MinhaFrame();
```

```
FlowLayout layout = new FlowLayout();
Container c = f.getContentPane();
c.setLayout(layout);
c.add(meuComponente)
...
```

CLASSE `BorderLayout`

- Organiza os componentes em cinco áreas no painel:
  Norte, Sul, Leste, Oeste e Centro.
- Padrão para os painéis de **JFrame**.

```
// 5 pixels de espacamento horizontal e
// 6 de espacamento vertical
// entre componentes.
JFrame f = new MinhaFrame();
BorderLayout b =  new BorderLayout(5,6);
Container c = f.getContentPane();
c.setLayout(b);
c.add(meuComponente,BorderLayout.SOUTH);
```



CLASSE `GridLayout`

- Divide um painel em linhas e colunas.
- Componentes são colocados no painel de modo a preencher cada posição da esquerda para direira, de cima para baixo.

```
private Container c = getContentPane();
private GridLayout grid;
// uma grade com 2 X 3, com espacamento 5 pixels X 5 pixels
```

```
// entre componentes
grid = new GridLayout (2, 3, 5, 5);
c.setLayout (grid);
for (int i = 0; i < NUM_BUT; i++) {
    b[i] = new JButton (...); b[i].addActionListener (...);
    c.add(b[i]);
}
```

12.6 Modelo de Eventos

- Quando um usuário interage com um componente da interface, um evento pode ser gerado.
- Eventos comuns:
  - mouse em movimento;
  - botão do mouse pressionado;
  - digitação de um campo de texto (ENTER);
  - abertura de uma janela;
  - seleção de um item de menu.
- Para processar um evento são necessárias duas tarefas:
  - o registro de um ouvinte de eventos;
  - a implementação de um tratador (handler) de eventos.

INTERFACE `ActionListener`

- Objetos ouvintes de eventos do tipo `ActionEvent` devem ser de classe que implementa `ActionListener`

```
interface ActionListener {
  public void actionPerformed (ActionEvent e)
}
```

- Métodos da classe `ActionEvent`:
  - `getSource( )`:
    retorna a referência do objeto originador do evento
  - `String getActionCommand( )` :
    retorna o string do comando associado com a ação, por exemplo, o rótulo de um botão

VISÃO GERAL DA MANIPULAÇÃO DE EVENTOS

- Um objeto ouvinte é instância de uma classe que implementa uma interface `EventListener`
- Uma origem de eventos é um objeto que pode registrar objetos ouvintes e a eles envia seus eventos

- A origem envia objetos eventos para todos os ouvintes dos eventos
- Os objetos ouvintes usam a informação do objeto evento recebido para determinar sua reação ao evento

EVENTOS E OUVINTES I

Interface	**Métodos**	**Parâmetro/ Acessadores**	**Origem**

`ActionListener`	`actionPerformed`	`ActionEvent`
		`getSource`
		`getActionCommand`
			`Button`
			`List`
			`MenuItem`
			`TextField`

Evento do tipo `ActionEvent` gerado por objeto:

- `Button` que recebe clique
- `List` que tem um item selecionado
- `MenuItem` que tem um item selecionado
- `TextField` que recebe um clique de ENTER

REGISTRO DO OUVINTE DE EVENTOS

- Cada objeto `x`, originador de eventos, do tipo **JComponent** faz referência a uma lista de tratadores de eventos a ele associados.
- O comando `x.addEventListener (eventListener)`
  associa um ouvinte (um tratador de eventos) chamado **eventListener** ao componente `x`.
  - Event acima pode ser: Action, Window, Mouse, MouseMotion, Key, etc
- Cada EventListener é uma interface (são 11)
- Quando um evento é disparado sobre algum objeto do tipo **JComponent**, sua lista de eventos é pesquisada em busca de um (ouvinte) tratador adequado.

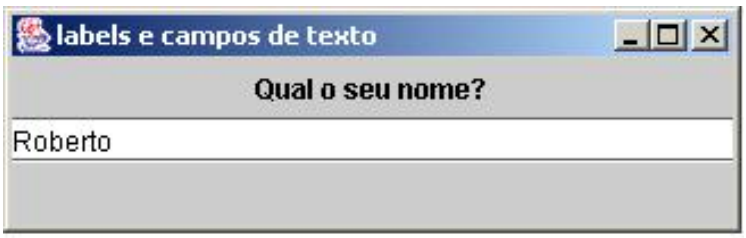
`JLabel` e `JTextField`

- Este programa mostra dois componentes **Swing** e um tratador de eventos.

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;
```

USO DE `Jlabel` e `JTextField`

```
>java UsaEntradaDeTexto
```



USO DE JLabel e JTextField

```
public class EntradaDeTexto extends JFrame {
   private class TratadorDeEventosDeTexto
                       implements ActionListener{...}
   private JTextField text = new JTextField (30);
   private JLabel label = new JLabel ("Qual o seu nome?");
   EntradaDeTexto() {
      super ("labels e campos de texto");
      Container c = getContentPane();
      c.setLayout (new FlowLayout() );
      c.add (label); c.add (text);
      text.addActionListener(new TratadorDeEventosDeTexto());
      setSize (325, 100);  setVisible(true);
   }
}
```

- classe interna para tratamento de evento implementada como uma classe aninhada da classe EntradaDeTexto

```
 private class TratadorDeEventosDeTexto implements
                       ActionListener {
     public void actionPerformed (ActionEvent e) {
         String s = "";
         if (e.getSource() == text)
```

```
                s = "Ola " + e.getActionCommand() + "!";
            JOptionPane.showMessageDialog (null, s);
        }
    }

public class UsaEntradaDeTexto {
   public static void main (String args[]) {
      EntradaDeTexto j = new EntradaDeTexto();
      j.addWindowListener (
          new WindowAdapter () {
              public void windowClosing (WindowEvent e) {
                  System.exit(0);
              }
          }
      );
   }
}
```

EVENTOS E OUVINTES II

Interface	**Métodos**	**Parâmetro/ Acessadores**	**Origem**

`WindowListener`	`windowClosing`
	`windowOpened`
	`windowIconified`
	`windowDeiconified`
	`windowClosed`
	`windowActivated`
	`windowDeactivated`
		`WindowEvent`
		`getWindow`
			`Window`

Eventos de Janela ocorrem quando a janela:

- é fechada com clique no X ou aberta;
- torna-se ativa com clique na janela ou torna-se inativa
- é minimizada (iconizada) com clique no *Box*
- é restaurada (desiconizada) com clique no ícone

CLASSES ADAPTADORAS

- Constituem uma alternativa às interfaces ouvintes de eventos.
- Permitem ao programador implementar apenas os métodos necessários.
- A abordagem de interface obriga o programador a implementar todos os métodos da interface.
- Classes adaptadoras implementam interfaces deixando os métodos com o corpo vazio.
- O programador estende uma classe adaptadora e redefine apenas os métodos que precisar.

CLASSE ADAPTADORA `WindowAdapter`

```
class WindowAdapter implements WindowListener {
    public void windowClosing(WindowEvent e) { }
    public void windowOpened(WindowEvent e) { }
    public void windowIconified(WindowEvent e) { }
    public void windowDeiconified(WindowEvent e) { }
    public void windowClosed (WindowEvent e) { }
    public void windowActivated (WindowEvent e) { }
    public void windowDeactivated(WindowEvent e) { }
}
```

CLASSES ADAPTADORAS DE EVENTOS

Classe Adaptadora	Interface Implementada
ComponentAdapter	**ComponentListener**
ContainerAdapter	**ContainerListener**
FocusAdapter	**FocusListener**
KeyAdapter	**KeyListener**
MouseAdapter	**MouseListener**
MouseMotionAdapter	**MouseMotionListener**
WindowAdapter	**WindowListener**

12.7 Botões de Comando

BOTÕES

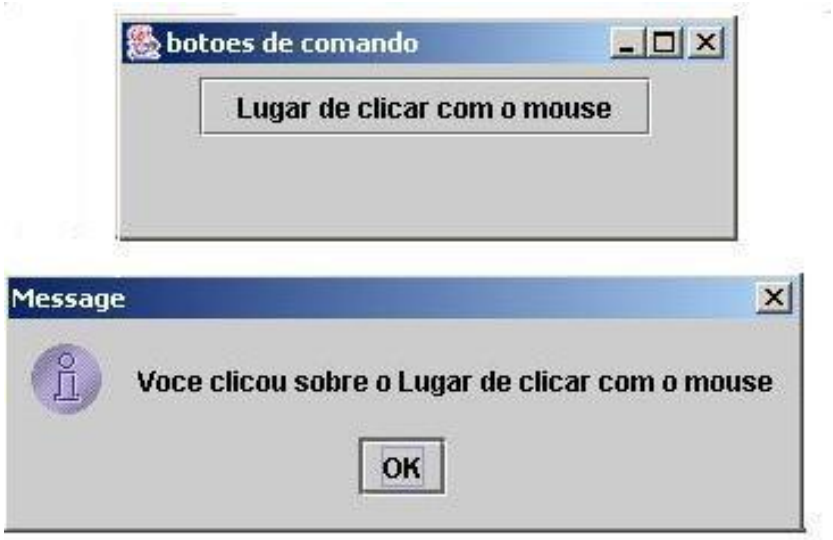
- Um *botão* é um componente que o usuário precisar clicar para acionar uma ação específica.
- Existem vários tipos de botões:
  - botões de comando:
    - `JButton`
  - botões de estado:
    - `JCheckBox`
    - `JRadioButton`

BOTÕES DE COMANDOS

- Botões de comando constituem um dos principais modos de interação entre o usuário e a aplicação.
- 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 forma.
- Um botão de comando gera um `ActionEvent` quando pressionado com o mouse.
- Os botões de comando podem ter palavras como rótulos ou figuras, as quais permitem a implementação de interfaces de uso bastante simples e intuitivo.

USO DE BOTÕES DE COMANDO

```
>java UsaExemploBotao
```



BOTÕES DE COMANDO

```
import java.awt.*;import java.awt.event.*;import javax.swing.*;
public class ExemploBotao extends JFrame {
    private class TratadorDeEventosDeBotao
                    implements ActionListener{ ... }
    private JButton botao =
        new JButton ("Lugar de clicar com o mouse");
    public ExemploBotao() {
        super ("botoes de comando");
        Container c = getContentPane();
        c.setLayout (new FlowLayout() );  c.add (botao);
        botao.addActionListener(new TratadorDeEventosDeBotao());
        setSize (275, 100);  setVisible(true);
    }
}

private class TratadorDeEventosDeBotao
                implements ActionListener {
    public void actionPerformed (ActionEvent e){
        JOptionPane.showMessageDialog (null,
           "Voce clicou sobre o " + e.getActionCommand() );
    }
}

import java.awt.*; import java.awt.event.*;
import javax.swing.*;
public class UsaExemploBotao {
```

```
    public static void main (String args[]){
        ExemploBotao j = new ExemploBotao();
        j.addWindowListener (
            new WindowAdapter () {
                public void windowClosing (WindowEvent e) {
                    System.exit(0);
                }
            }
        );
    }
}
```

BOTÕES DE ESTADO

- Botões de estado são caracterizados por valores de estado: ativado/desativado, verdadeiro/falso, etc.
- O pacote **Swing** disponibiliza dois tipos de botões de estado:
  - **JCheckBox**
  - **JRadioButton**
- Botões de estado geralmente são utilizados para selecionar uma opção (**JRadioButton**) ou várias opções (**JCheckBox**).
- As classes **JRadioButton** e **JCheckBox** são subclasses de **JToggleButton**.

EVENTOS E OUVINTES III

Interface	**Método**	**Parâmetro/ Acessadores**	**Origem**

`ItemListener`	`itemState-Changed`	`ItemEvent`
		`getItem`
		`getItemSelectable`
		`getStateChange`
			`CheckBox`
			`CheckBoxMenuItem`
			`Choice`
			`List`

- `ItemEvent` ocorre quando o usuário faz uma seleção num conjunto de caixas de seleção ou itens de uma lista
- `ItemEvent.SELECTED`: indica que item foi selecionado
- `ItemEvent.DESELECTED`: indica que item foi desselecionado
- `int getStateChange()`: retorna o tipo de mudança de estado (`SELECTED` ou `DESELECTED`)

12.8 Botões de Estado

USO DE `JCheckBox`

```
>java UsaDiversasOpcoes
```

JCheckBox

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class DiversasOpcoes extends JFrame {
  private class ControladorDeOpcoes implements ItemListener{...}
  private JCheckBox bold = new JCheckBox("Bold");
  private JCheckBox italic = new JCheckBox ("Italic");
  private JTextField t =
     new JTextField ("Olha como que as letras mudam", 20);
  public DiversasOpcoes () { ... }
}

public DiversasOpcoes () {
      Container c = getContentPane();
      ControladorDeOpcoes cdo = new ControladorDeOpcoes();
      c.setLayout (new FlowLayout());
      t.setFont(new Font ("TimesRoman", Font.PLAIN, 14));
      c.add(t);   c.add (bold);   c.add(italic);
      bold.addItemListener (cdo);
      italic.addItemListener (cdo);
      setSize (275, 100);
      setVisible(true);
}

private class ControladorDeOpcoes implements ItemListener {
  private int valBold = Font.PLAIN, valItalic = Font.PLAIN;
  public void itemStateChanged (ItemEvent e){
    if (e.getSource() == bold)
       if (e.getStateChange() == ItemEvent.SELECTED)
            valBold = Font.BOLD;
       else valBold = Font.PLAIN;
    if (e.getSource() == italic)
       if (e.getStateChange()==ItemEvent.SELECTED)
            valItalic = Font.ITALIC;
       else valItalic = Font.PLAIN;
```

```
    t.setFont(new Font("TimesRoman",valBold + valItalic, 14));
    t.repaint();
}}

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class UsaDiversasOpcoes {
  public static void main (String args[]) {
     DiversasOpcoes j = new DiversasOpcoes();
     j.addWindowListener (
         new WindowAdapter () {
             public void windowClosing (WindowEvent e) {
                    System.exit(0);
             }
         }
     ); }
}
```

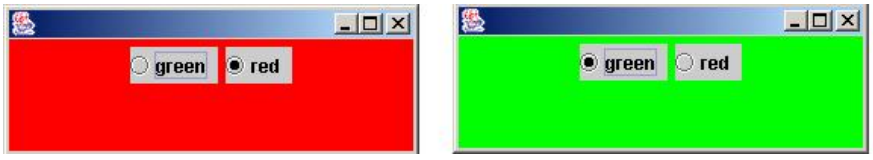
12.9 Botões de Rádio

SELEÇÃO DE OPÇÕES MUTUAMENTE EXCLUSIVAS

- **JRadioButton**s são usados quando deseja-se oferecer ao usuário um conjunto de opções dentre as quais somente uma pode ser escolhida
- Um conjunto de **JRadioButton**s mutuamente exclusivos deve ser agrupado por meio da classe **ButtonGroup**.
- Cada um dos botões que mutuamente se excluem deve ser adicionado a uma instância da classe **ButtonGroup**.

USO DE JRadioButton

```
>java UsaUmaOpcao
```



JRadioButton

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class UmaOpcao extends JFrame {
  private class TratadorDeRBut implements ItemListener {...}
  private TratadorDeRBut controlador = new TratadorDeRBut();
  private JRadioButton green = new JRadioButton ("green", false);
  private JRadioButton red   = new JRadioButton ("red", true);
  private JButtonGroup radioGroup = new ButtonGroup();
  private Container c = getContentPane();

  public UmaOpcao() {...}
}

public UmaOpcao() {
        c.setBackground (Color.red);
        c.add (green);
        c.add (red);

        red.addItemListener (controlador);
        green.addItemListener (controlador);

        radioGroup.add(red);
        radioGroup.add(green);

        setSize (275, 100);
        setVisible(true);
}

private class TratadorDeRBut implements ItemListener {
    public void itemStateChanged (ItemEvent e) {
        if (e.getSource() == red)
            c.setBackground (Color.red);
        else if (e.getSource() == green)
            c.setBackground (Color.green);
    }
}

import java.awt.*; import java.awt.event.*;
import javax.swing.*;
public class UsaUmaOpcao {
   public static void main (String args[]) {
      UmaOpcao j = new UmaOpcao();
      j.addWindowListener (
          new WindowAdapter () {
              public void windowClosing (WindowEvent e) {
                  System.exit(0);
              }
          }
      );
```

```
  }
}
```

12.10 Caixas de Combinação

SELEÇÃO DE ITENS DE LISTA

- Estes componentes fornecem listas de itens para o usuário fazer uma escolha.
- **JComboBox**:
  - permite ao usuário escolher 1 item de uma lista
  - geram eventos percebidos por **itemListener**.
- **JList**:
  - pode ser usada para implementar listas de múltiplas seleções.
  - geram eventos percebidos por **listSelectionListener**.

CAIXA DE COMBINAÇÃO (JComboBox)

- Itens da lista são identificados por índices, sendo 0 (zero) o primeiro
- Primeiro item adicionado à lista aparece como item selecionado quando JComboBox for exibida
- Para cada seleção são gerados dois eventos: um para o item desselecionado e outro para o selecionado

```
private String names[] = {"desenho1", "desenho2, "desenho3"};
Container c = getContentPane();

private JComboBox images = new JComboBox (names);
images.setMaximumRowCount(12);
c.add (images);
```

OPERAÇÕES DE JComboBox

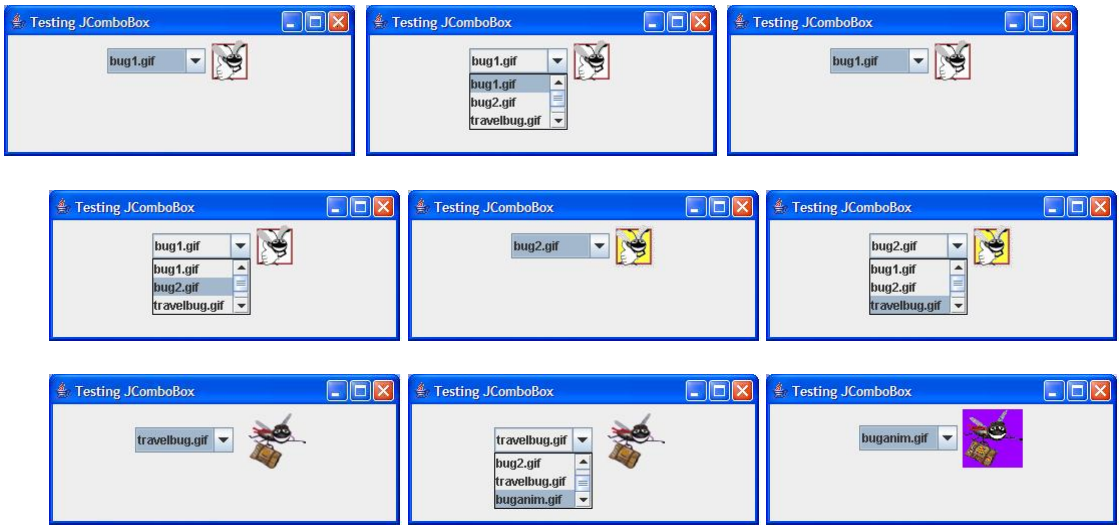
- `JComboBox(String[ ] names )`:
- `setMaximumRowCount( int n )`:
  define número de linhas visíveis da caixa de combinação. Se necessário barras de rolagem serão automaticamente incluídas
- `int getSelectedIndex( )`:
  índice dos itens inicia-se com 0

JComboBoxFame

```
import javax.swing.JFrame;

public class ComboBoxTest {
   public static void main( String args[] ) {
```

```
        ComboBoxFrame c = new ComboBoxFrame();
        c.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        c.setSize( 350, 150 );
        c.setVisible( true );
 }
}
```

JComboBoxFrame



Importações Necessárias à Classe `ComboBoxFrame`

```
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;
```

Classe `ComboBoxFrame`

```
public class ComboBoxFrame extends JFrame {
   private JComboBox comboBox;
   private JLabel label;
   private String names[] = {
      "bug1.gif","bug2.gif","travelbug.gif","buganim.gif"
   };
   private Icon icons[] = {
      new ImageIcon( names[ 0 ]),
      new ImageIcon( names[ 1 ]),
      new ImageIcon( names[ 2 ]),
      new ImageIcon( names[ 3 ])
```

```
   };
   public ComboBoxFrame( ){ ... }
}
```

Construtora de `ComboBoxFrame`

```
public ComboBoxFrame( ){
   super("Testing JComboBox");
   setLayout( new FlowLayout( ) );

   comboBox = new JComboBox( names );
   comboBox.setMaximumRowCount( 3 );//exibe três linhas

   "ADICIONA OUVINTE ItemListener A comboBox" ;

   add(comboBox );
   label = new JLabel(icons[ 0 ]); //exibe primeiro ícone
   add(label);
}
```

ADICIONA OUVINTE `ItemListener` A comboBox

```
comboBox.addItemListener(
   new ItemListener( ) {
       public void itemStateChanged(ItemEvent event){
         if (event.getStateChange( ) == ItemEvent.SELECTED) {
            label.setIcon(icons[comboBox.getSelectedIndex( )]);
         }
       }
   }
)
```

12.11 Listas

OPERAÇÕES DE JList

- `JList( Object[ ] itemNames)`:
  mostra na lista os elementos de `itemNames`, normalmente convertidos com `toString`.
- `setVisibleRowCount( int n )`
- `int setSelectionMode(int mode)`:
  define o modo de seleção de itens, que pode ser `SINGLE_SELECTION`, `SINGLE_INTERVAL_SELECTION` ou `MULTIPLE_INTERVAL_SELECTION`
- `setListData(Object[ ] itens )`:
  configura os itens dados, por default convertidos com `toString`, no Jlist corrente
- `setCellRenderer(ListCellRenderer renderer)`:
  delega ao método `renderer.getListCellRendererComponent(...)`, no lugar de `toString`, a produção de cada um dos itens da JList

- `int getSelectedIndex( ) :`
- `int[ ] getSelectedIndices( ) :`
- `Object getSelectedValue( ):`
  retorna o rótulo do item selecionado
- `Object[ ] getSelectedValues( ):`
  retorna os rótulos dos itens selecionados
- `setFixedCellWidth( int w )`
- `setFixedCellHeight( int h )`

VALORES DO SelectionMode

- `static int ListSelectionModel.SINGLE_SELECTION`:
  permite selecionar de um único item de cada vez
- `static int ListSelectionModel.SINGLE_INTERVAL_SELECTION` permite selecionar um intervalo contíguo de itens com o auxílio da teclas SHIFT
- `static int ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`:
  permite seleção de intervalos contíguo com o auxílio da tecla CTL

Seleção de Uma Cor de Uma JList

```
import javax.swing.JFrame;
public class ListTest {
  public static void main( String args[] )  {
    ListFrame listFrame = new ListFrame();
    listFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    listFrame.setSize( 350, 150 );
    listFrame.setVisible( true );
  }
}
```


```
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.ListSelectionModel;

public class ListFrame extends JFrame {
   private JList colorJList;
   private final String colorNames[] = { "Black", "Blue",
      "Cyan", "Dark Gray", "Gray", "Green", "Light Gray",
      "Magenta","Orange", "Pink", "Red", "White", "Yellow"
```

```
   };
   private final Color colors[] = { Color.BLACK, Color.BLUE,
      Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
      Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
      Color.RED, Color.WHITE, Color.YELLOW
   };
   public ListFrame(){ ... }
}
```

CONSTRUTORA DE ListFrame

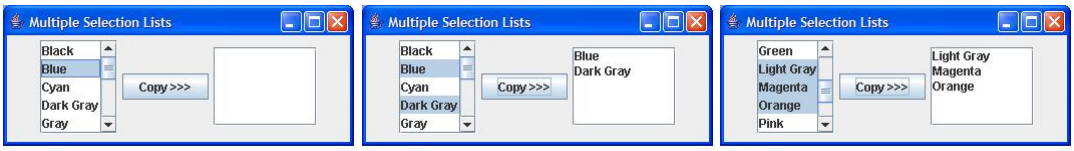
```
public ListFrame() {
   super( "List Test" );  setLayout(new FlowLayout( ));

   colorJList = new JList( colorNames );
   colorJList.setVisibleRowCount( 5 );

   // não permite múltiplas seleções
   colorJList.setSelectionMode(
                   ListSelectionModel.SINGLE_SELECTION);

   "ADICIONA OUVINTE ListSelectionListener A colorJList";

   add( new JScrollPane( colorJList ) );
}
```

ADICIONA OUVINTE ListSelectionListener A colorJList

```
colorJList.addListSelectionListener(
   new ListSelectionListener() {
       public void valueChanged(ListSelectionEvent event){
          getContentPane().setBackground(
                  colors[colorJList.getSelectedIndex()] );
       }
   }
);
```

SELEÇÃO DE VÁRIAS CORES DE UMA LISTA

```
import javax.swing.JFrame;
public class MultipleSelectionTest {
   public static void main( String args[] ) {
      MultipleSelectionFrame m = new MultipleSelectionFrame();
      m.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE );
      m.setSize( 350, 140 );
      m.setVisible( true );
   }
}
```




```
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;
```

CLASSE MultipleSelectionFrame

```
public class MultipleSelectionFrame {
   private JList colorJList;
   private JList copyJList;
   private JButton copyJButton;

   private final String colorNames[] = { "Black", "Blue",
      "Cyan", "Dark Gray", "Gray", "Green", "Light Gray",
      "Magenta", "Orange", "Pink", "Red", "White", "Yellow"
   };

   public MultipleSelectionFrame() { ... }
}
```

CONSTRUTORA DE MultipleSelectionFrame

```
public MultipleSelectionFrame() {
   super( "Multiple Selection Lists" );
   setLayout( new FlowLayout() );

   "CONSTRÓI LISTA DE SELEÇÃO colorJList";
   add( new JScrollPane( colorJList ) );

   copyJButton = new JButton( "Copy >>>" );
   "ADICIONA OUVINTE ActionListener A copyJButton";
   add( copyJButton );

   "CONSTRÓI LISTA copyList";
   add( new JScrollPane( copyJList ) );
}
```

CONSTRÓI LISTA DE SELEÇÃO colorJList

```
colorJList = new JList( colorNames );
```

```
colorJList.setVisibleRowCount( 5 );

colorJList.setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

ADICIONA OUVINTE ActionListener A copyJButton

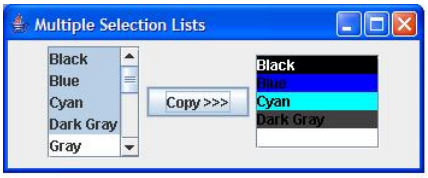
```
copyJButton.addActionListener(
   new ActionListener() {
       public void actionPerformed(ActionEvent event) {
          copyJList.setListData(colorJList.getSelectedValues());
       }
   }
);
```

CONSTRÓI LISTA copyJList

```
copyJList = new JList();

copyJList.setVisibleRowCount( 5 );

copyJList.setFixedCellWidth( 100 );

copyJList.setFixedCellHeight( 15 );

copyJList.setSelectionMode(
       ListSelectionModel.SINGLE_INTERVAL_SELECTION);
```

SELEÇÃO DE VÁRIAS CORES DE UMA LISTA

```
import javax.swing.JFrame;
public class MultipleSelectionTest {
   public static void main( String args[] ) {
      MultipleSelectionFrame m = new MultipleSelectionFrame();
      m.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE );
      m.setSize( 350, 140 );
      m.setVisible( true );
   }
}
```




```
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;
import javax.swing.JFrame;
```

```
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;
import javax.swing.ListCellRenderer
```

CLASSE MultipleSelectionFrame

```
public class MultipleSelectionFrame {
   private JList colorJList, JList copyJList;
   private JButton copyJButton;
   private final String colorNames[] = { "Black", "Blue",
      "Cyan", "Dark Gray", "Gray", "Green", "Light Gray",
      "Magenta", "Orange", "Pink", "Red", "White", "Yellow"
   };
   private final Color colors[] = { Color.black, Color.blue,
      Color.cyan, Color.darkGray, Color.gray, Color.green,
      Color.lightGray, Color.magenta, Color.orange, Color.pink,
      Color.red, Color.white, Color.yellow
   };
   public MultipleSelectionFrame() { ... }
}
```

CONSTRUTORA DE MultipleSelectionFrame

```
public MultipleSelectionFrame() {
   super( "Multiple Selection Lists" );
   setLayout( new FlowLayout() );

   "CONSTRÓI LISTA DE SELEÇÃO colorJList";
   add( new JScrollPane( colorJList ) );

   copyJButton = new JButton( "Copy >>>" );
   "ADICIONA OUVINTE ActionListener A copyJButton";
   add( copyJButton );

   "CONSTRÓI LISTA copyList";
   add( new JScrollPane( copyJList ) );
}
```

CONSTRÓI LISTA DE SELEÇÃO colorJList

```
colorJList = new JList( colorNames );

colorJList.setVisibleRowCount( 5 );

colorJList.setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

ADICIONA OUVINTE ActionListener A copyJButton

```
copyJButton.addActionListener(
   new ActionListener() {
       public void actionPerformed(ActionEvent event) {
          copyJList.setListData(colorJList.getSelectedValues());
       }
   }
);
```

CONSTRÓI LISTA copyJList

```
copyJList = new JList();

copyJList.setVisibleRowCount( 5 );

copyJList.setFixedCellWidth( 100 );

copyJList.setFixedCellHeight( 15 );

copyJList.setSelectionMode(
       ListSelectionModel.SINGLE_INTERVAL_SELECTION);

copyJList.setCellRenderer(new MyRenderer(colors));
```

Produtor de Itens de JList (Versão I)

```
interface ListCellRenderer {
   public Component getListCellRendererComponent(
          JList list, Object value, int index,
          boolean isSelected, boolean cellHasFocus);
}
class MyRenderer extends JLabel implements ListCellRenderer {
   private Color[ ] colors;
   public MyCellRenderer(Color[ ] colors) {
      this.colors = colors;
   }
   public Component getListCellRendererComponent(
          JList list, Object value, int index,
          boolean isSelected, boolean cellHasFocus){...}
}
```

Produtor de Itens de JList (Versão I)

```
public Component getListCellRendererComponent(
       JList list, Object value,  int index,
       boolean isSelected, boolean cellHasFocus)  {
   setText(value.toString()); index = index%colors.length;
   setBackground(colors[index]); setOpaque(true);
   setForeground((colors[index] == Color.black)?
                                   Color.white : Color.black);
```

```
   return this;
}
```

- Retorna o componente a ser exibido como o item de posição `index` da `JList` `list`
- Automaticamente chamado para cada item de `list`
- Na falta de objeto `ListCellRenderer` associado ao `JList`, o componente exibido na posição `index` do `JList` é `value.toString()`

Produtor de Itens de JList (Versão II)

```
interface ListCellRenderer {
   public Component getListCellRendererComponent(
          JList list, Object value, int index,
          boolean isSelected, boolean cellHasFocus);
}
class MyRenderer extends implements ListCellRenderer {
   private Color[ ] colors;
   public MyCellRenderer(Color[ ] colors) {
      this.colors = colors;
   }
   public Component getListCellRendererComponent(
          JList list, Object value, int index,
          boolean isSelected, boolean cellHasFocus){...}
}
```

Produtor de Itens de JList (Versão II)

```
public Component getListCellRendererComponent(
       JList list, Object value,  int index,
       boolean isSelected, boolean cellHasFocus)  {
   index = index%colors.length;
   JButton item = new JButton( );
   item.setText(value.toString());
   item.setOpaque(true);
   item.setBackground(colors[index]);
   item.setForeground((colors[index] == Color.black)?
                                  Color.white : Color.black);
   return item;
}
```

12.12 Painéis

CLASSE `Jpanel`

- Painéis são usados como uma base onde os componentes gráficos podem ser fixados, inclusive outros painéis.
- Painéis são criados com **JPanel** – uma subclasse de **JComponent**:

  - cria-se uma classe que estende `JPanel`
  - sobrepõe-se o método `paintComponent(Graphics g)`
  - adicionam-se componentes ao painel com `add(Component c)`

- `paintComponent` não deve ser chamado diretamente. Use `repaint( )`
- O primeiro comando de `paintComponent` deve ser sempre a chamada `super.paintComponent(g)`
- `paint(Graphics g)` desenha no `JFrame`
- `paintComponent(Graphics g)` desenha no `JPanel`, que pode ter sido adicionado a um `JFrame`

MÉTODOS DA CLASSE `JPanel`

- `JPanel( )`:
  cria painel com `FlowLayout`
- `void setLayout(LayoutManager layout)`:
  associa leiaute ao painel
- `JPanel(LayoutManager layout)`:
  cria painel com o leiaute indicado
- `add(Component c)`:
  Adiciona o componente gráfico `c` no painel.

USANDO PAINÉIS

```
>java UsaPanelDemo
```


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class PanelDemo extends JFrame {
    private JPanel  buttonPanel = new JPanel();
    private JButton buttons[]   = new JButton[5];

    public PanelDemo () { ...}

}



public PanelDemo () {
    super ("Demonstrando Paineis");
    Container c = getContentPane();
    buttonPanel.setLayout(new GridLayout (2,3));
    for (int i = 0; i < buttons.length; i++) {
         buttons[i] = new JButton ("Button " + (i + 1));
         buttonPanel.add (buttons[i]);
    }
    c.add (buttonPanel, BorderLayout.SOUTH);
    setSize (425, 150);
    setVisible(true);
}



public class UsaPanelDemo {
   public static void main (String args[]){
     PanelDemo j = new PanelDemo();
     j.addWindowListener (
         new WindowAdapter() {
             public void windowClosing (WindowEvent e) {
                    System.exit(0);
             }
         }
     );
   }
}
```

JPanel PERSONALIZADO

```
>java UsaPainelPersonalizado
```

ESTENDENDO `JPanel`

```
import java.awt.*; import java.swing.event.*;
import javax.swing.*;

public class PainelPersonalizado extends JPanel {
    public void paintComponent (Graphics g) {
        super.paintComponent (g);
        g.fillOval(50, 10, 60, 60);
    }
}

import java.awt.*; import java.swing.event.*;
import javax.swing.*;
public TestePainelPersonalizado extends JFrame {
    public TestePainelPersonalizado () {
        super ("Teste do painel personalizado");
        PainelPersonalizado meuPainel =
                    new PainelPersonalizado();
        meuPainel.setBackground (Color.yellow);
        Container c = getContentPane();
        c.add(meuPainel, BorderLayout.SOUTH);
        setSize(300, 150); setVisible(true);
    }
}

import java.awt.*; import java.awt.event.*;
import javax.swing.*;
public class UsaPainelPersonalizado {
  public static void main (String args[]) {
```

```
    TestePainelPersonalizado j = new TestePainelPersonalizado();
        j.addWindowListener (
            new WindowAdapter () {
               public void windowClosing (WindowEvent e) {
                   System.exit(0);
               }
            }
         );
   }
}
```

REMOÇÃO E INCLUSÃO DE COMPONENTES

```
   >java UsaRemoveComponente
```


```
import java.awt.*; mport java.awt.event.*;import javax.swing.*;
public class UsaRemoveComponente{
     public static void main (String args[]){
        RemoveComponente j = new RemoveComponente();
        j.addWindowListener (
            new WindowAdapter() {
                public void windowClosing (WindowEvent e) {
                    System.exit(0);
                }
            }
        );
    }

}

import java.awt.*;import java.awt.event.*;import javax.swing.*;
public class RemoveComponente extends JFrame
                                  implements ActionListener{
    private JPanel  buttonPanel = new JPanel();
    private JButton buttons[ ]  = 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 RemoveComponente( ){...}
    public void actionPerformed(ActionEvent e) {...}
}

public RemoveComponente( ) {
    super ("Adição e Remoção de Componentes");
    buttonPanel.setLayout(new GridLayout (2,3));
    for (int i = 0; i < buttons.length; i++) {
          buttons[i] = new JButton ("Button " + i);
          buttonPanel.add(buttons[i]);
          buttonPanel.add(buttons[i]); // somente um inserido
    }
    "Adiciona Componentes na Janela";
    "Associa Ouvinte a Botões";
    setSize (425, 150);
    show();
}
```

- `"Adiciona Componentes na Janela"`:

```
Container c = getContentPane();
c.add(remocao2,BorderLayout.WEST);
c.add(remocao4,BorderLayout.EAST);
c.add(adicao2,BorderLayout.NORTH);
c.add(adicao4,BorderLayout.CENTER);
c.add(buttonPanel, BorderLayout.SOUTH);
```

- `"Associa Ouvinte a Botões"`:

```
remocao2.addActionListener(this);
remocao4.addActionListener(this);
adicao2.addActionListener(this);
adicao4.addActionListener(this);
```

```
public void actionPerformed(ActionEvent e) {
       if  (e.getSource()== remocao2)
               buttonPanel.remove(buttons[2]);
       else if (e.getSource()== remocao4)
               buttonPanel.remove(buttons[4]);
       else if (e.getSource()== adicao2)
               buttonPanel.add(buttons[2]);
       else if (e.getSource()== adicao4)
               buttonPanel.add(buttons[4]);

       buttonPanel.repaint();
}
```

12.13 Eventos de Mouse

EVENTOS DE MOUSE

- Eventos de mouse são capturados por objetos do tipo das interfaces: `MouseListener`, `MouseMotionListener` e `MouseWheelListener`.
- Podem ser disparados sobre qualquer componente GUI que deriva de **java.awt.Component**.
- São disparados por ações do mouse: clicar, arrastar, etc.
- Passam para a classe manipuladora de eventos as coordenadas `(x,y)` onde aconteceu o evento.
- Java disponibiliza duas interfaces e sete métodos virtuais para tratamento de eventos de mouse.

INTERFACES MANIPULADORAS DE EVENTOS DE MOUSE

- **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.
- `MouseWheelListener` captura eventos gerados pela rotação da roda do mouse.
- Os métodos de **MouseListener** e **MouseMotionListener** são chamados quando têm-se as três condições:
  1. o mouse gera o evento
  2. o mouse interage com um componente
  3. objetos ouvintes apropriados foram registrados no componente

EVENTOS E OUVINTES IV

Interface	Método	Parâmetro/ Acessadores	Origem

`MouseListener`	`mousePressed`
	`mouseReleased`
	`mouseEntered`
	`mouseExited`
	`mouseClicked`
		`MouseEvent`
		`getClickCount`
		`getX , getY`
		`getPoint`
		`getButton`
		`getModifiers`
		`getMouseModifierText`
		`isPopupTrigger`
			`Component`

`MouseMotion-Listener`	`mouseDragged` `mouseMoved`	`MouseEvent`	`Component`
`MouseWheel-Listener`	`mouseWheelMoved`	`MouseWheelEvent` `...`	`Component`

CLASSE MouseEvent

- **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. Valor retornado é uma das constantes: `NOBUTTON, BUTTON1, BUTTON2, BUTTON3`
- **int getModifier()**:
  retorna um int descrevendo as teclas modificadoras (Shift, Ctl+Shift, etc) ativas durante evento
- **String getMouseModifierText(int)**:
  retorna um String descrevendo as teclas modificadoras (Shift, Ctl+Shift, etc) contidas no int
- **boolean isPopUpTrigger( )**:
  retorna `true` se o evento de mouse deve causar o aparecimento de um menu pop-up

MÉTODOS DE MouseListener

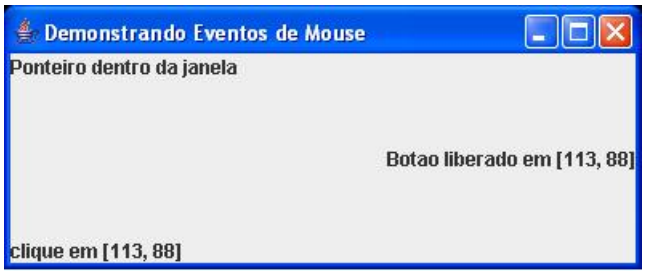
- **public void mousePressed (MouseEvent e)**
  - botão do mouse pressionado com cursor sobre um componente.
- **public void mouseClicked (MouseEvent e)**
  - botão do mouse pressionado e liberado sobre um componente sem movimentação do cursor (click).
- **public void mouseReleased (MouseEvent e)**
  - botão do mouse liberado depois de ser pressionado.
    Sempre precedido por **mousePressed**.
- **public void mouseEntered (MouseEvent e)**
  - chamado quando cursor do mouse entra nos limites de um componente.
- **public void mouseExited (MouseEvent e)**
  - chamado quando cursor do mouse deixa os limites de um componente.

MÉTODOS DE MouseMotionListener

- **public void mouseDragged (MouseEvent e)**
  - botão do mouse é pressionado e o cursor é movido. Sempre precedido por **mousePressed**.
- **public void mouseMoved (MouseEvent e)**
  - mouse movido com o cursor sobre um componente.

DEMONSTRANDO EVENTO DE MOUSE

```
>java UsaEvtMouse
```




```
import java.awt.*;import java.awt.event.*;import javax.swing.*;
public class EvtMouseDemo extends JFrame
             implements MouseListener, MouseMotionListener {

  private JLabel status1 = new JLabel();
  private JLabel status2 = new JLabel();
  private JLabel status3 = new JLabel();

  private Container c = getContentPane();

  "CONSTRUTORA DE EvtMouseDemo";

  "Operações de tratamento de evento";
}
```

CONSTRUTORA DE EvtMouseDemo

```
  public EvtMouseDemo () {
      super ("Demonstrando Eventos de Mouse");

      c.add(status1,BorderLayout.SOUTH);
      c.add(status2,BorderLayout.NORTH);
      c.add(status3,BorderLayout.EAST);

      addMouseListener (this); addMouseMotionListener (this);

      setSize(275, 100); setVisible(true);
  }

}
```

OPERAÇÕES DE TRATAMENTO DE EVENTOS

```
public void mouseClicked (MouseEvent e) {
   status1.setText
    ("clique em [" + e.getX() + ", " + e.getY() + "]");
}

public void mousePressed (MouseEvent e) {
```

```
   status3.setText
    ("Botao pressionado em [" + e.getX()+ ", " + e.getY()+ "]");
}

public void mouseReleased (MouseEvent e) {
   status3.setText
     ("Botao liberado em [" + e.getX() + ", " + e.getY() + "]");
}

public void mouseEntered (MouseEvent e) {
   status2.setText ("Ponteiro dentro da janela");
}

public void mouseExited (MouseEvent e) {
   status2.setText ("Ponteiro fora da janela");
}

public void mouseDragged (MouseEvent e) {
   status1.setText ("Arrastando em [" +
             e.getX() + ", " + e.getY() + "]");
}

public void mouseMoved (MouseEvent e) {
   status1.setText ("Movendo em [" +
             e.getX() + ", " + e.getY() + "]");
}
```

DEMONSTRANDO EVENTO DE MOUSE: PRINCIPAL

```
public class UsaEvtMouse {
    public static void main (String args[]){
       EvtMouseDemo j = new EvtMouseDemo ();

       j.addWindowListener (
           new WindowAdapter() {
               public void windowClosing (WindowEvent e) {
                    System.exit(0);
               }
           }
       );
    }
}
```

UM PROGRAMA DE PINTURA

```
   >java Painter
```


```
public class Painter extends JFrame {
    private int x = -10; y = -10;
    public  Painter () {
        super ("Uma tela eletronica");
        getContentPane().add (
            new Label ("Arraste o mouse para desenhar"),
                                          BorderLayout.SOUTH);
        addMouseMotionListener (...);
        setSize (300, 150);  setVisible(true);
    }
    public void paint (Graphics g){ g.fillOval(x,y,4,4); }
    public static void main (String args[]){...}
}

addMouseMotionListener (
     new MouseMotionAdapter() {
            public void mouseDragged (MouseEvent e) {
                    x = e.getX();
                    y = e.getY();
                    repaint();
            }
      }
);

public static void main (String args[]){
     Painter j = new Painter ();
     j.addWindowListener (
         new WindowAdapter() {
             public void windowClosing (WindowEvent e){
                System.exit(0);
             }
        }
     );
}
```

12.14 Eventos de Teclado

EVENTOS DE TECLADO

- Eventos de teclado são gerados quando as teclas são pressionadas e liberadas.

- `KeyListener` é a interface ouvinte de eventos de teclado.
- `KeyListener` tem três métodos:
  - `keyPressed` é chamado em resposta ao pressionamento de qualquer tecla.
  - `keyTyped` é chamado em resposta ao pressionamento de tecla que não é uma *tecla de ação*: *Home, Page Up*, etc.
  - `keyReleased` é chamado quando a tecla é liberada, depois do evento `keyPressed` ou `keyTyped`.

CÓDIGOS VIRTUAIS DE TECLAS

- Método `getKeyCode()` informa o código virtual da tecla
- Método `getKeyChar()` informa o caractere digitado
- Códigos virtuais, definidos pela classe `KeyEVent`, têm prefixo `VK_`.
  Por exemplo, `VK_A`, `VK_B`, `VK_SHIFT`, ect

EXEMPLO DE EVENTOS DE TECLADO

- 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"`: 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"`: chamado `keyTyped` com `"a"`
  3. Liberada a tecla A: chamado `keyReleased` com `VK_A`

EVENTOS E OUVINTES V

Interface	**Método**	**Parâmetro/ Acessadores**	**Origem**

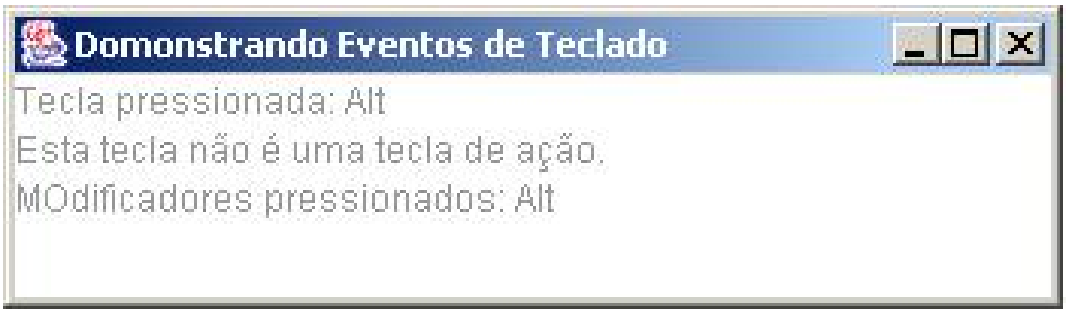
`KeyListener`	`keyPressed`
	`keyReleased`
	`keyTyped`
		`KeyEvent`
		`getKeyChar`
		`getKeyCode`
		`getKeyModifiersText`
		`getKeyText`
		`isActionKey`
			`Component`

MÉTODOS DE KeyEvent

- `char getKeyChar()`:
  devolve o caractere digitado
- `int getKeyCode()`:
  devolve o código virtual do caractere digitado
- `String getKeyText(int codTecla)`:
  devolve texto descrevendo o código virtual da tecla.
  Ex: `getKeyText(KeyEvent.VK_END)` devolve `"END"`
- `int getKeyModifiers( )`:
  devolve a máscara do modificador associado ao evento
- `String getKeyModifiersText(int modificadores)`:
  devolve descrição das tecla modificadoras (SHIFT, CTRL, CTRL+SHIFT)
  Parâmetro obtido pelo método `getModifiers()`
- `boolean isAltDown()`:
  idem para tecla ALT
- `boolean isControlDown()`:
  idem para tecla CONTROL
- `boolean isShiftDown()`:
  idem para tecla SHIFT

DEMONSTRANDO EVENTOS DE TECLADO

```
>java UsaTecladoDemo
```




```
import javax.swing.*; import java.awt.*;
import java.awt.event.*;
public class UsaTecladoDemo extends JFrame
                  implements KeyListener {
  private String line1 = "", line2 = "", line3 = "";
  private JTextArea textArea = new JTextArea (10, 15);
```

```
  public TecladoDemo () {...}
  public void keyPressed (KeyEvent e) {...}
  public void keyReleased (KeyEvent e) {...}
  public void keyTyped (KeyEvent e) {...}
  private void ativaLinhas2e3 (KeyEvent e) {...}
  public static void main (String args[]){...}
}

public UsaTecladoDemo () {
     super ("Demonstrando Eventos de Teclado");
     textArea.setText ("Aperte alguma tecla no teclado");
     textArea.setEnabled (false);
     // permite que o frame processe os eventos de teclado
     addKeyListener (this);
     getContentPane().add(textArea);
     setSize (350, 100);
     setVisible(true);
}

public void keyPressed (KeyEvent e) {
   line1 = "Tecla pressionada: " + e.getKeyText(e.getKeyCode());
   ativaLinhas2e3(e);
}
public void keyReleased (KeyEvent e) {
   line1 = "Tecla liberada: " + e.getKeyText(e.getKeyCode());
   ativaLinhas2e3(e);
}
public void keyTyped (KeyEvent e) {
  line1 = "Tecla pressionada: " + e.getKeyChar();
  ativaLinhas2e3(e);
}

private void ativaLinhas2e3 (KeyEvent e) {
  line2 = "Esta tecla " + (e.isActionKey() ? "": "não ") +
                                       "é uma tecla de ação.";

  String temp = e.getKeyModifiersText (e.getModifiers());
  line3 = "MOdificadores pressionados: " +
                (temp.equals("") ? "nenhum" : temp);

  textArea.setText(line1 + "\n" + line2 + "\n" + line3 + "\n");
}

public static void main (String args[]){
      UsaTecladoDemo j = new UsaTecladoDemo();
      j.addWindowListener (
         new WindowAdapter() {
             public void windowClosing (WindowEvent e) {
```

```
                System.exit(0);
            }
        }
    );
}
```

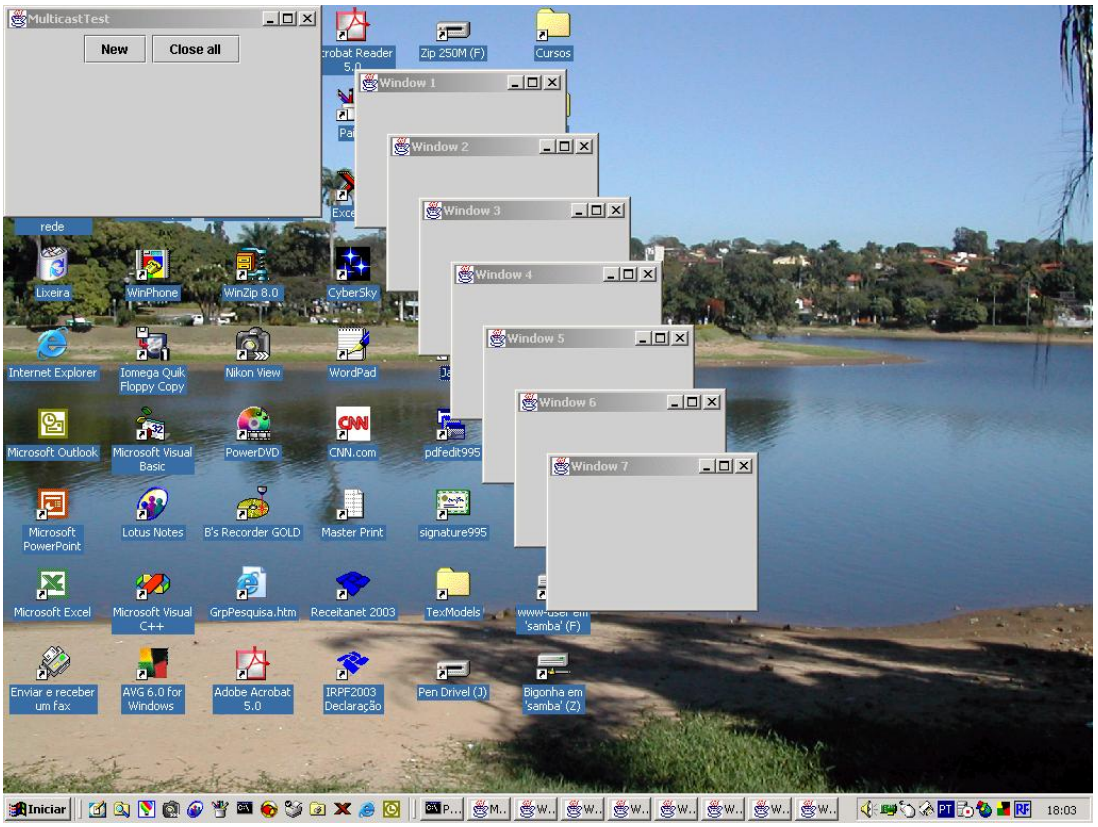
MULTIDIFUSÃO DE EVENTOS

- As origens de eventos do AWT suportam um modelo de multidifusão para os ouvintes registrados
- Um mesmo evento pode ser enviado para mais de um ouvinte
- Multidifusão é útil quando em evento desperta o interesse em muitas partes do programa

- Uma janela com dois botões de comando:
  - Botão `New`: comanda a criação de uma nova janela
  - Botão `Close All`: fecha todas as janelas

```
>java MulticastTest
```


```
>java MulticastTest
```




```
import java.awt.*; import java.awt.event.*;
import javax.swing.*;
class MulticastPanel extends JPanel implements ActionListener {
   private static int counter=0;private JButton closeAllButton;
   public static void setCounter(int c){counter = c;}
   public MulticastPanel( ){
      JButton newButton = new JButton("New");
      add(newButton);
      newButton.addActionListener(this);
      closeAllButton = new JButton("Close all");
      add(closeAllButton);
   }
  public void actionPerformed(ActionEvent evt){ ... }
}

public void actionPerformed(ActionEvent evt){
      SimpleFrame f = new SimpleFrame();
      f.setTitle("Window " + counter);
      f.setSize(200, 150);
      f.setLocation(30 * counter, 30 * counter);
      f.show();
      closeAllButton.addActionListener(f);
}
```

```
class SimpleFrame extends JFrame implements ActionListener {
   public void actionPerformed(ActionEvent evt){
      MulticastPanel.setCounter(0);
      dispose();
   }
}


class MulticastFrame extends JFrame{
 public MulticastFrame() {
   setTitle("MulticastTest"); setSize(300, 200);
   addWindowListener(new WindowAdapter( ){
    public void windowClosing(WindowEvent e){System.exit(0);}});
   Container contentPane = getContentPane();
   contentPane.add(new MulticastPanel());
 }
}
public class UsaMulticast {
 public static void main(String[] args) {
    JFrame frame = new MulticastFrame(); frame.show();
 }
}
```

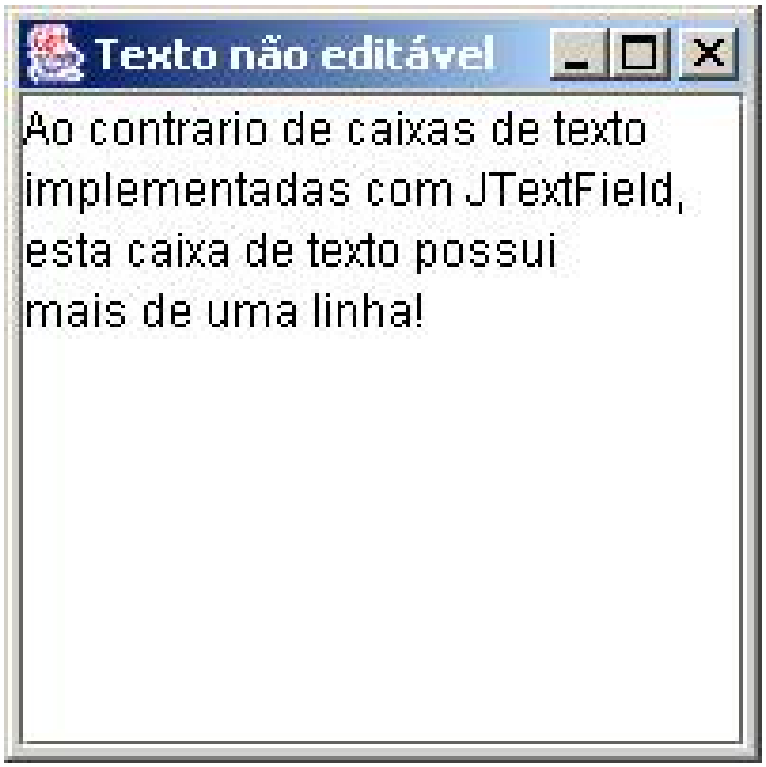
12.15 Áreas de Texto

COMPONENTE JTextArea

- **JTextArea** fornece uma área para manipulação de múltiplas linhas de texto.
- **JTextArea**, assim como **JTextField**, herda de **JTextComponent**.
- Um componente **JTextArea** pode ser ou não editável pelo usuário.
- É possível fornecer barras de rolagem a uma **JTextArea** associando-a com um objeto **JScrollPane**.
- Métodos de `JTextArea`:
  - `setEditable(boolean b)`
  - `JTextArea(String t, int largura, int altura)`

`JTextArea` COM BARRA DE ROLAGEM

```
>java UsaTextAreaDemo
```



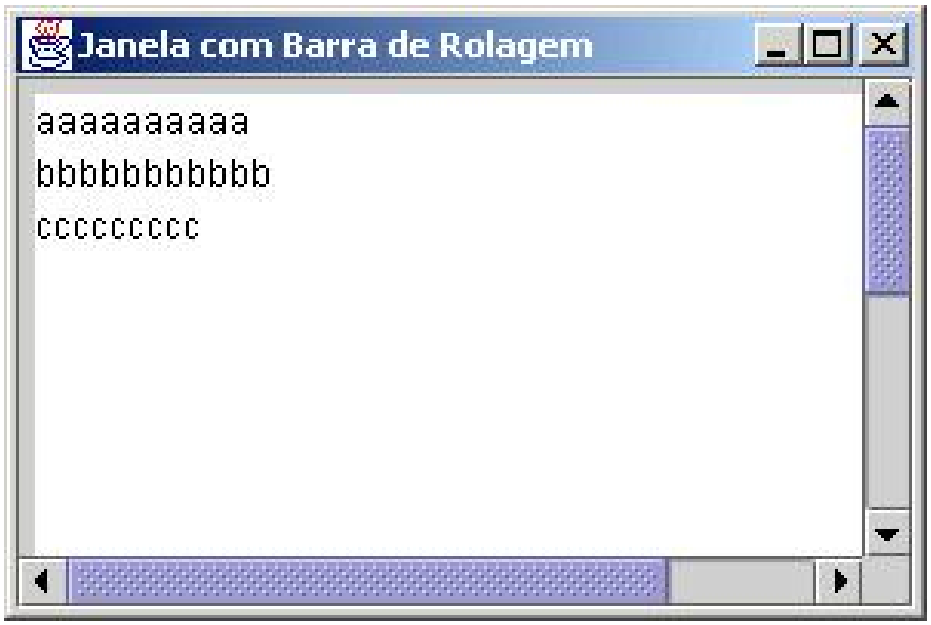
JTextArea COM BARRA DE ROLAGEM

```
import java.awt.*;
import java.swing.event.*;
import javax.swing.*;

public class TextAreaDemo extends JFrame {
    private JTextArea tArea;
    public TextAreaDemo() {....}
}

    public TextAreaDemo() {
        super ("Texto não editável");
        Container c = getContentPane();
        String s = "Ao contrario de caixas de texto\n" +
                   "implementadas com JTextField,\n" +
                   "esta caixa de texto possui\n" +
                   "mais de uma linha!\n";
        tArea = new JTextArea (s, 10, 15);
        tArea.setEditable (false);
        c.add (new JScrollPane(tArea));
        setSize(200, 200);
        setVisible(true);
```

```
    }

import java.awt.*; import java.awt.event.*;
import javax.swing.*;
public class UsaTextAreaDemo {
  public static void main (String args[]) {
    TextAreaDemo j = new TextAreaDemo();
    j.addWindowListener (
       new WindowAdapter () {
           public void windowClosing (WindowEvent e) {
                  System.exit(0);
              }
          }
      );
  }
}
```

JANELA COM BARRAS DE ROLAGEM

```
>java UsaJanelaComScrollBar
```




```
public class UsaJanelaComScrollBar {
   public static void main(String [ ] args) {
      JFrame janela = new JanelaComScrollBar( );
      janela.addWindowListener(
         new WindowAdapter( ){
            public void windowClosing(WindowEvent e){
                System.exit(0);
```

```
            }
         }
      );
   }
}
```

```
import javax.swing.*; import java.awt.*;
import java.awt.event.*;
class JanelaComScrollBar extends JFrame {
   public JanelaComScrollBar( ) {
      setTitle("Janela com Barra de Rolagem"); setSize(300,200);
      Container c = getContentPane();
      JPanel j = new JPanel( );
      c.add(new JScrollPane(j));
      String s ="aaaaaaaaaa\nbbbbbbbbbbb\nccccccccc";
      JTextArea t = new JTextArea(s,20,30);
      j.add(t);
      setVisible(true);
  }
}
```

12.16 Barras Deslizantes

COMPONENTE JSlider

- O componente **JSlider** também chamado de *Controle Deslizante* permite ao usuário selecionar a partir de um intervalo de valores inteiros.
- **JSlider** são como caixas de rolagens graduadas com marcas de medida.
- As marcas de escala de um **JSlider** são altamente personalizáveis.
- **JSlider** suportam tanto eventos de mouse como de teclado.
- **JSlider** de orientação horizontal têm seu valor mínimo no lado esquerdo.

MÉTODOS DE JSlider

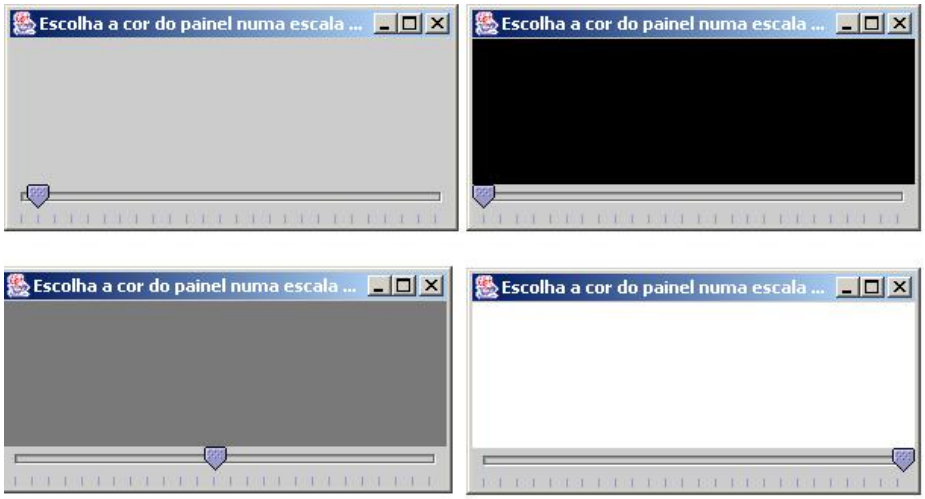
- `JSlider(int orient, int min, int max, int init)`:
  um dos construtores da classe: *orientação, valor mínimo, valor máximo* e *valor inicial* do componente.
- `setMajorTickSpacing(int)`:
  determina o intervalo entre duas marcas visíveis do componente **JSlider**.
- `setPaintTicks (boolean)`:
  determina se as marcas de graduação estarão visíveis ou não.
- `getValue()`:
  retorna o valor corrente do componente **JSlider**.

EVENTOS E OUVINTES VI

Interface	Método	Parâmetro/ Acessadores	Origem
`ChangeListener`	`stateChanged`	`ChangeEvent` `getSource`	`JSlider`

DEMONSTRADOR DE JSlider

```
>java UsaSliderDemo
```



UM JPanel DE COR CONFIGURÁVEL

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class SliderDemo extends JFrame {
  private JSlider tomDeCinza =
     new JSlider (SwingConstants.HORIZONTAL, 0, 255, 10);
  private JPanel pC = new JPanel(); // painel cinza
  public SliderDemo() { ... }
}

public SliderDemo() {
    super ("Escolha a cor do painel numa escala de cinza");
    final Container c = getContentPane();
    c.add (pC, BorderLayout.CENTER);
    tomDeCinza.setMajorTickSpacing(10);
    tomDeCinza.setPaintTicks(true);
    tomDeCinza.addChangeListener (
       new ChangeListener() {
```

```
          public void stateChanged (ChangeEvent e) {
             int cor = tomDeCinza.getValue();
             pC.setBackground(new Color (cor, cor, cor));
       }} );
    c.add(tomDeCinza); setSize(300,150); setVisible(true);
}
```

USANDO O DEMONSTRADOR DE JSlider

```
import java.awt.*; import java.awt.event.*;
import javax.swing.*;
public class UsaSliderDemo {
  public static void main (String args[]){
     SliderDemo j = new SliderDemo();
     j.addWindowListener (
        new WindowAdapter () {
            public void windowClosing(WindowEvent e){
                    System.exit(0);
            }
        }
    );
  }
}
```

12.17 Barras de Menus, Menus e SubMenus

MENU DE OPÇÕES

- *Menus* são um meio de organizar as funcionalidades de um aplicativo de maneira prática e natural.
- Em GUI's Swing, apenas instâncias das classes que fornecem o método **setJMenuBar** podem usar menus.
- As classes utilizadas para definir menus são:
  - **JMenuBar**
  - **JMenuItem**
  - **JCheckBoxMenuItem**
  - **JRadioButtonMenuItem**
  - **JMenu**

CLASSES DE MENU

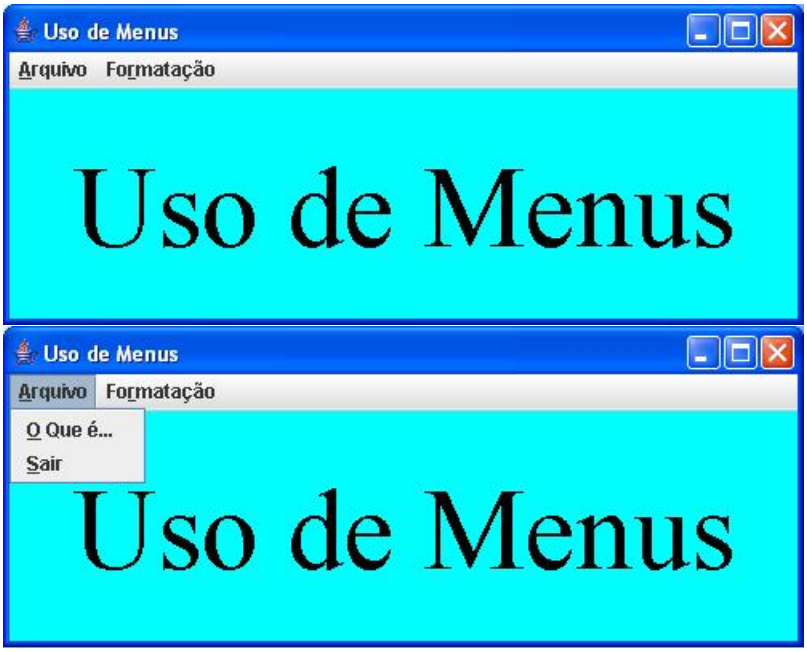
- A classe **JMenuBar** contém métodos para gerenciar uma *barra de menus*.
- A classe **JMenuItem** contém métodos para gerenciar *itens de menus*.
- A classe **JMenu** contém os métodos para gerenciar *menus*
- A classe **JCheckBoxMenuItem** contém métodos para gerenciar itens de menu que podem ser ativados ou desativados.

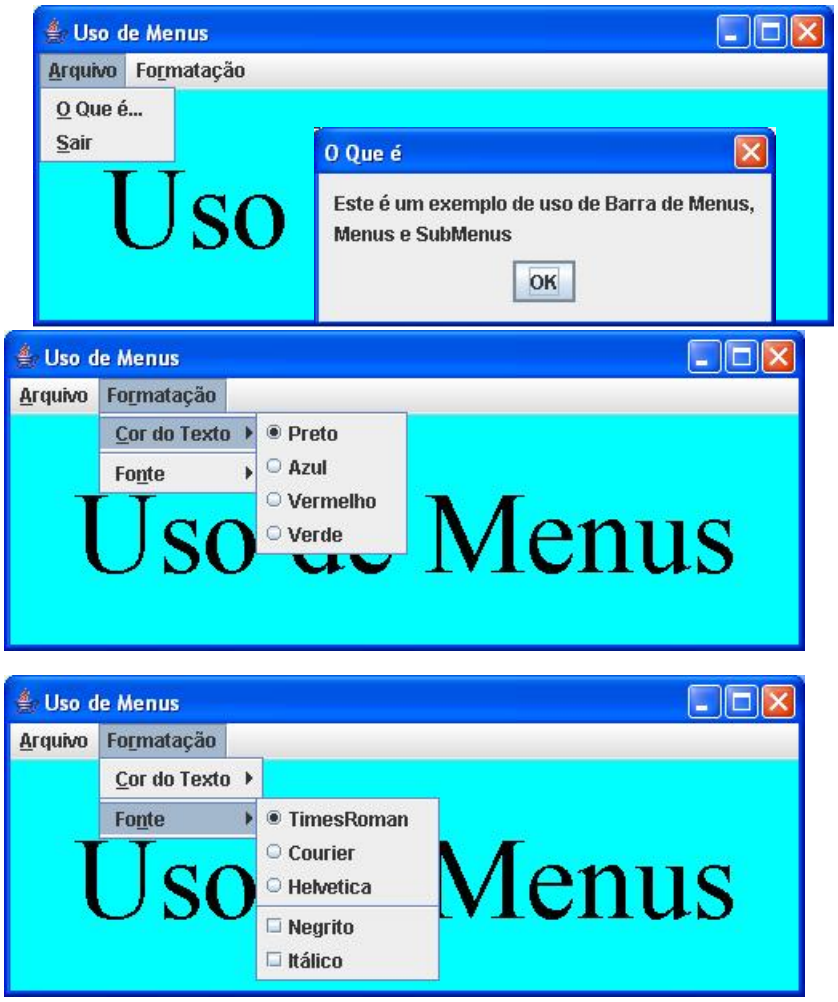
- A classe **JRadioButtonMenuItem** contém métodos que gerenciam um conjunto de itens de menu de modo que em um dado momento apenas um item pode estar ativo.

BARRA DE MENUS, MENUS E SUBMENUS

```
import javax.swing.*;
import java.awt.event.*; import java.awt.*;
public class UsaMenu {
   public static void main( String args[] ) {
      MenuTest m = new MenuTest();
      m.addWindowListener(
         new WindowAdapter() {
            public void windowClosing( WindowEvent e ) {
               System.exit( 0 );
            }
         }
      );
   }
}
```

```
>java UsaMenu
```






```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class MenuTest extends JFrame {
   private Color colorValues[] =
      { Color.black, Color.blue, Color.red, Color.green };
   private JRadioButtonMenuItem colorItems[], fonts[];
   private ButtonGroup fontGroup, colorGroup;
   private JCheckBoxMenuItem styleItems[];
   private JLabel display;
   private int style;
   public MenuTest( ) { ... }
}
```

CONSTRUTOR DE MenuTest

```
public MenuTest() {
   super( "Uso de Menus" );
```

```
   JMenuBar bar = new JMenuBar();  setJMenuBar( bar );

   "CONSTRÓI MENU fileMenu";   bar.add(fileMenu);
   "CONSTRÓI MENU formatMenu"; bar.add( formatMenu);

   Container c = getContentPane();
   c.setBackground( Color.cyan );
   "CONSTRÓI RÓTULO display";
   c.add( display, BorderLayout.CENTER );

   setSize( 500, 200 );  show();
}
```

CONSTRÓI RÓTULO display

```
   display = new JLabel("Uso de Menus",SwingConstants.CENTER);

   display.setForeground( colorValues[ 0 ] );

   display.setFont(new Font( "TimesRoman", Font.PLAIN, 72 ) );
```

CONSTRÓI MENU fileMenu

```
JMenu fileMenu = new JMenu( "Arquivo" );
fileMenu.setMnemonic( 'A' );
file Menu.setToolTipText("Decida-se logo");

"CONSTRÓI MENU O Que é";
fileMenu.add( aboutItem );

"CONSTRÓI MENU exitMenu";
fileMenu.add(exitItem );
```

CONSTRÓI MENU O Que é

```
JMenuItem aboutItem = new JMenuItem( "O Que é..." );
aboutItem.setMnemonic( 'O' );

aboutItem.addActionListener(
  new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
      JOptionPane.showMessageDialog( MenuTest.this,
       "Este é um exemplo de uso de Barra de Menus,\n" +
       "Menus e SubMenus","O Que é",JOptionPane.PLAIN_MESSAGE);
    }
  }
```

```
);
```

CONSTRÓI MENU exitMenu

```
JMenuItem exitItem = new JMenuItem( "Sair" );
exitItem.setMnemonic( 'S' );

exitItem.addActionListener(
   new ActionListener() {
       public void actionPerformed(ActionEvent e){
           System.exit( 0 );
       }
   }
);
```

CONSTRÓI MENU formatMenu

```
JMenu formatMenu = new JMenu( "Formatação" );
formatMenu.setMnemonic( 'r' );
formatMenu.setToolTipText("Formata o rótulo");

"CONSTRÓI SUBMENU colorMenu";
formatMenu.add( colorMenu );

formatMenu.addSeparator();

"CONSTRÓI SUBMENU fontMenu";
formatMenu.add( fontMenu );
```

CONSTRÓI SUBMENU colorMenu

```
String colors[] ={ "Preto", "Azul", "Vermelho", "Verde" };
JMenu colorMenu = new JMenu( "Cor do Texto" );
colorMenu.setMnemonic( 'C' );
colorItems = new JRadioButtonMenuItem[ colors.length ];
colorGroup = new ButtonGroup();
ItemHandler itemHandler = new ItemHandler();
for (int i = 0; i < colors.length; i++) {
     colorItems[ i ] = new JRadioButtonMenuItem(colors[i]);
     colorMenu.add( colorItems[ i ] );
     colorGroup.add( colorItems[ i ] );
     colorItems[ i ].addActionListener( itemHandler );
}
colorItems[ 0 ].setSelected( true );
```

CONSTRÓI SUBMENU fontMenu

```
JMenu fontMenu = new JMenu( "Fonte" );
fontMenu.setMnemonic( 'n' );

"ADICIONA ITENS DE FONTES AO fontMenu";

fontMenu.addSeparator( );

"ADICIONA ITENS DE ESTILOS AO fontMenu";
```

ADICIONA ITENS DE FONTES AO fontMenu

```
String fontNames[] = { "TimesRoman", "Courier", "Helvetica" };
fonts = new JRadioButtonMenuItem[ fontNames.length ];
fontGroup = new ButtonGroup();
fontMenu.addSeparator();
for ( int i = 0; i < fonts.length; i++ ) {
    fonts[ i ] = new JRadioButtonMenuItem( fontNames[ i ] );
    fontMenu.add( fonts[ i ] );
    fontGroup.add( fonts[ i ] );
    fonts[ i ].addActionListener( itemHandler );
}
fonts[ 0 ].setSelected( true );
```

ADICIONA ITENS DE ESTILOS AO fontMenu

```
String styleNames[] = { "Negrito", "Itálico" };
styleItems = new JCheckBoxMenuItem[ styleNames.length ];
StyleHandler styleHandler = new StyleHandler();

for ( int i = 0; i < styleNames.length; i++ ) {
    styleItems[i] = new JCheckBoxMenuItem(styleNames[i]);
    fontMenu.add( styleItems[i]);
    styleItems[i].addItemListener(styleHandler);
}
```

TRATADOR DE SELEÇÃO DE ITEM DE COR

```
class ItemHandler implements ActionListener {
   public void actionPerformed( ActionEvent e ) {
      for ( int i = 0; i < colorItems.length; i++ )
          if ( colorItems[ i ].isSelected() ) {
              display.setForeground(colorValues[i]); break;
          }
      for ( int i = 0; i < fonts.length; i++ )
          if (e.getSource( ) == fonts[i]) {
              display.setFont( new Font(
```

```
                fonts[i].getText(),style,72)); break;
          }
      repaint();
   }
}
```

TRATADOR DE SELEÇÃO DE ITEM DE ESTILO

```
class StyleHandler implements ItemListener {
   public void itemStateChanged( ItemEvent e ) {
      style = 0;
      if ( styleItems[ 0 ].isSelected() )
           style += Font.BOLD;
      if ( styleItems[ 1 ].isSelected() )
           style += Font.ITALIC;
      display.setFont( new Font(
           display.getFont().getName(), style, 72 ) );
      repaint();
   }
}
```

12.18 Menus Sensíveis ao Contexto

MENUS SENSÍVEIS AO CONTEXTO - JPopupMenus

- Um *Menu sensível ao contexto* ou *pop-up* revela opções específicas para o componente gráfico para o qual foi ativado.
- Menus *pop-up* são ativados por eventos denominados *eventos de gatilho pop-up.*
- O disparador de PopUpMenu depende da plataforma
- Por exemplo, clique com botão direito sobre um componente é o gatilho pop-up no Windows
- Método `isPopupTrigger( )` da classe `MouseEvent` informa se evento foi causado por ativação de menu `pop-up`
- Método `mpu.show(componente, x, y)` faz a exibição nas coordenadas `(x,y)` do menu `pop-up mpu`

UTILIZANDO JPopupMenus

```
class UsaPopupMenu {
  private JPopupMenu menu = new JPopupMenu( );

  "construir cada item de menu e adicioná-lo ao menu pop-up;
   determinar 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";

  addMouseListener (...);
  ...
}
```

```
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() );
        }
    }
);
```

12.19 Janelas Múltiplas

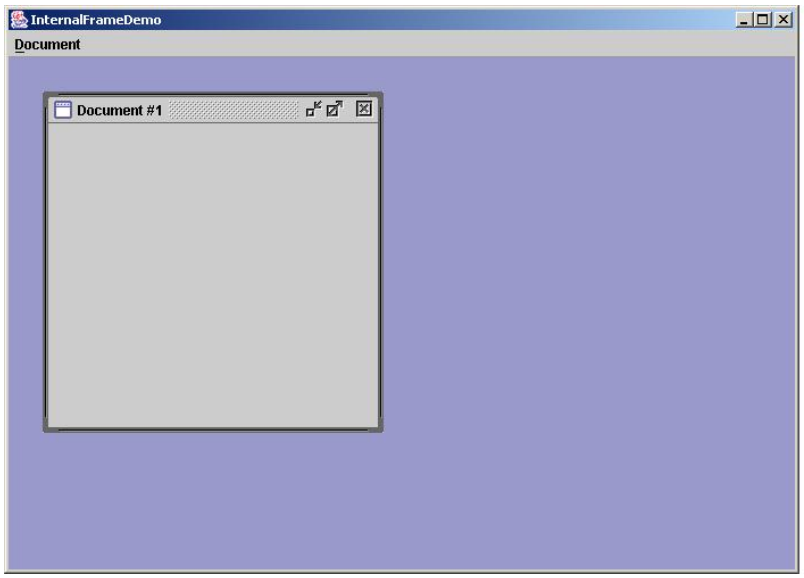
JDesktopPane e JInternalFrame

- Uma **Interface de Múltiplos Documentos** (`multiple document interface - MDI`) consiste em uma janela principal contendo outras janelas.
- A janela principal é frequentemente chamada *Janela Mãe*.
- As janelas internas são chamadas *Janelas Filhas*.
- Esta interface permite gerenciar vários documentos que estão sendo processados em paralelo.
- As classes **JDesktopPane** e **JInternalFrame** do Swing fornecem suporte para criar interfaces de múltiplos documentos.

GERÊNCIA DE MÚLTIPLOS DOCUMENTOS

```
public class UsaInternalFrame {
  public static void main(String[] args) {
     ExternalFrame frame = new ExternalFrame( );
     frame.setVisible(true);
  }
}
```

```
   >javaUsa UsaInternalFrame
```



GERÊNCIA DE MÚLTIPLOS DOCUMENTOS

```
import javax.swing.JInternalFrame;
import java.awt.event.*;
import java.awt.*;

public class InternalFrame extends JInternalFrame {
    static int openFrameCount = 0;
    static final int xOffset = 30, yOffset = 30;

    public InternalFrame() {...}

}
```

```
public InternalFrame() {
   super("Document #" + (++openFrameCount),
          true, //a frame pode ter seu tamanho modificado
          true, //a frame pode ser fechada.
          true, //a frame pode ser maximizada.
          true);//a frame pode ser minimizada.
   // Define o tamanho inicial da frame.
   setSize(300,300);
   // Define a localizacao inicial da nova frame.
   setLocation(xOffset*openFrameCount,
               yOffset*openFrameCount);
}
```

```
import javax.swing.JInternalFrame;
import javax.swing.JDesktopPane;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
```

```
import javax.swing.JMenuBar;
import javax.swing.JFrame;
import java.awt.event.*; import java.awt.*;
public class ExtenalFrame extends JFrame {
      JDesktopPane desktop;
      public ExternalFrame {...}
      protected JMenuBar createMenuBar( ) {...}
      protected void createFrame( ) {...}
}

public ExternalFrame() {
   super("InternalFrameDemo");
   // Make the big window be indented 50 pixels
   // from each edge of the screen.
   int inset = 50;
   Dimension screenSize =
         Toolkit.getDefaultToolkit().getScreenSize();
   setBounds(inset, inset, screenSize.width - inset*2,
             screenSize.height-inset*2);
   "Adiciona ouvinte para fechar janela";
   "Set up the GUI";
}
```

ADICIONA OUVINTE PARA FECHAR JANELA

```
addWindowListener(
       new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
             System.exit(0);
          }
       }
)
```

SET UP THE GUI

```
 desktop = new JDesktopPane();
 createFrame(); //Create first window
 setContentPane(desktop);
 setJMenuBar(createMenuBar());
 //Make dragging faster:
 desktop.putClientProperty(
               "JDesktopPane.dragMode", "outline");

protected JMenuBar createMenuBar() {
   JMenuBar menuBar = new JMenuBar();
   JMenu menu = new JMenu("Document");
   menu.setMnemonic(KeyEvent.VK_D);
   JMenuItem menuItem = new JMenuItem("New");
```

```
   menuItem.setMnemonic(KeyEvent.VK_N);
   menuItem.addActionListener(new ActionListener(){...});
   menu.add(menuItem);
   menuBar.add(menu);
   return menuBar;
}
```

```
menuItem.addActionListener(
     new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                createFrame();
            }
     }
);
```

```
 protected void createFrame() {
    InternalFrame frame = new InternalFrame();
    frame.setVisible(true); //necessary as of kestrel
    desktop.add(frame);
    try {
            frame.setSelected(true);
    } catch (java.beans.PropertyVetoException e) {}
 }
```

12.20 Aparência e Comportamento

APARÊNCIA E COMPORTAMENTO

- Programas Java que utilizam apenas classes definidas no pacote AWT têm a aparência e o comportamento da plataforma em que o programa executa.
- Os componentes GUI em cada plataforma têm aparências diferentes que podem requer quantidades diferentes de espaço para serem exibidas. Isso pode alterar o leiaute e o alinhamento dos componentes GUI.
- Os componentes GUI em cada plataforma têm funcionalidades padrão diferentes.
- Há componentes GUI do Swing que eliminam estas questões, fornecendo funcionalidade uniforme entre plataformas.

- O conjunto de propriedades como cor e textura que determinam o aspecto geral de qualquer componente em uma certa plataforma é denominado *look and feel*.
- Uma mesma plataforma pode possuir diferentes conjuntos *look and feel* instalados, mas em um dado momento apenas um estará sendo usado.
- A classe **UIManager** gerencia os conjuntos *look and feel* de uma plataforma.
- Objetos da classe **UIManager.LookAndFeelInfo** têm informações sobre os *look and feel* instalados no sistema.

Look and Feel - MÉTODOS RELACIONADOS

- `LookAndFeelInfo[ ] UIManager.getInstaledLookAndFeels()`:
  retorna a lista de *look and feel* instalados no sistema.

- `UIManager.setLookAndFeel(String lfClassname)`:
  faz o *look and feel* de nome `lfClassName` ser o corrente no sistema.

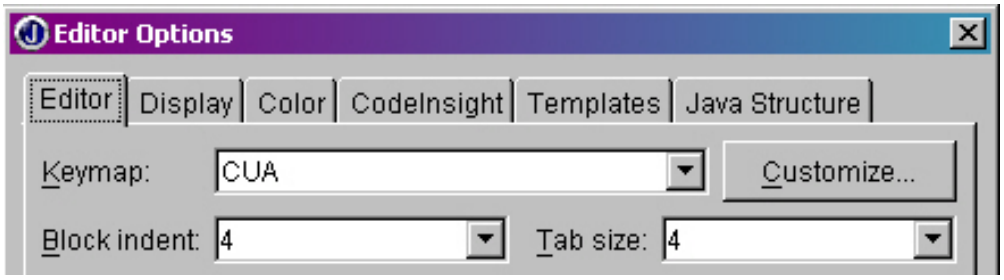
MUDANDO A APARÊNCIA DE UM APLICATIVO

- A aparência do aplicativo pode ser mudada via chamadas a função mudaVisual. Ex.: mudaVisual(2), etc.

```
...
UIManager.LookAndFeelInfo looks[] =
                    UIManager.getInstalledLookAndFeels();
...
private void mudaVisual (int value) {
   ...
   UIManager.setLookAndFeel(looks[value].getClassName());
   ...
}
```

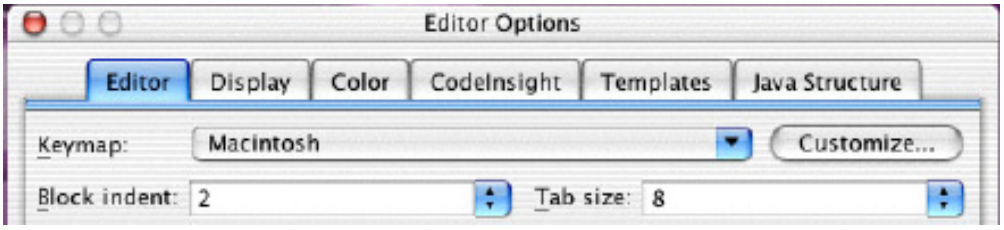
LOOK AND FEEL WINDOWS

```
UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```



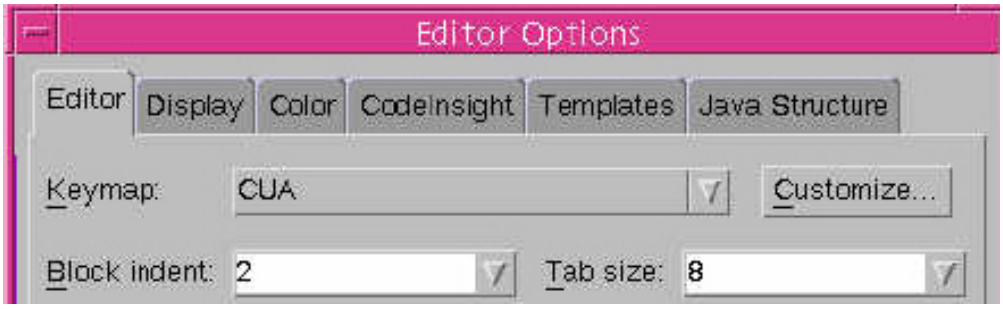
LOOK AND FEEL MAC

```
UIManager.setLookAndFeel(
          "");
```



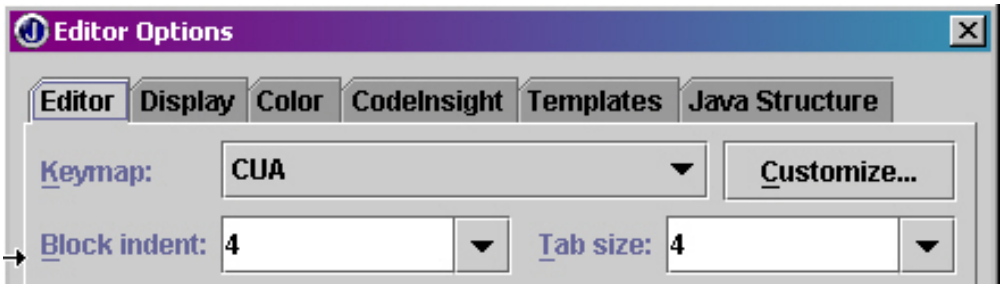
LOOK AND FEEL MOTIF

```
UIManager.setLookAndFeel(
      "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```



LOOK AND FEEL METAL

```
UIManager.setLookAndFeel(
          "javax.swing.plaf.metal.MetalLookAndFeel");
```



12.21 Exercícios

12.22 Notas Bibliográficas

Capítulo 13

Threads

13.1 Criação de *Thread*

LINHAS CONTROLE DE EXECUÇÃO

- Linhas de controle (*threads*) permitem que o programa execute mais de uma tarefa de cada vez.
- No caso de haver apenas um processador disponível a execução das linhas é entremeada.
- No caso de mais de um processador, pode haver paralelismo real entre as linhas.
- O número de processadores disponíveis não deve ser considerado para a correção de um programa multilinha (*Multithreading*).
- Em Java, linhas de controle são representados por meio de objetos.

CRIAÇÃO DE LINHAS: MÉTODO I

- Estender `java.lang.Thread`, sobrepondo `run()`
- O método `run()` de `Thread` nada faz

```
class T extends Thread {
   public void run( ) { ...}
   ...
}
```

- Criar objetos da linha e iniciar sua execução com `start()`

```
public class B {
  public static void main(String[] a) {
    T linha1 = new T();  // cria linha
    T linha2 = new T();  // cria outra linha
    linha1.start();  // dispara linha1
    ...
    linha2.start(); // dispara linha 2
    ....
  }
}
```

EXEMPLO: DOIS PROCESSOS PARALELOS

```
class PingPong extends Thread {
  String palavra; int tempo;
  PingPong(String texto, int espera) {
      palavra = texto;  tempo = espera;
  }
  public void run() {
    try {while (true) {
              System.out.print(palavra + " ");
              sleep(tempo);
        }
    }catch (InterruptedException e) { return; }
  }
}

class Jogo {

  public static void main(String[] args) {
    Thread t1 = new PingPong("Ping", 33);
    Thread t2 = new PingPong("PONG",100);
    t1.start(); t2.start();
  }

}
```

CRIAÇÃO DE LINHAS: MÉTODO II

- A partir da Interface `java.lang.Runnable`

  ```
  public interface Runnable {
    public abstract void run();
  }
  ```

- Implementar interface `java.lang.Runable`, definindo `run()`:

  ```
  class R extends X implements Runnable {
    public void run() { ... }
        ...
  }
  ```

- Quando um objeto que implementa `Runnable` é usado na criação de uma `thread`, o disparo da `thread` leva a execução de `run()`.

- Assim, declara-se uma classe que implementa `Runnable`:

  ```
  class R extends X implements Runnable {
    public void run() { ... }
        ...
  }
  ```

CRIAÇÃO DE LINHAS II ...

- Criam-se objetos da linha e inicia sua execução com o método `start()` de `Thread`:

```
class B {
  public static void main(String[] a) {
    Thread linha = new Thread(new R()); // cria linha
    linha.start(); ... // inicia linha
    new Thread(new R()).start(); cria e inicia outra linha
}
```

- Uma linha termina quando seu `run` termina.

(RE)DEFINIÇÃO DE `run()`

```
class Exemplo {
   public static void main(String[] a) {
      ExtnOfThread t1 = new ExtnOfThread();
      t1.start();
      Thread t2 = new Thread (new ImplOfRunnable());
      t2.start();
   }
}

class ExtnOfThread extends Thread {
   public void run() {
     System.out.println("extension of Thread running");
     setPriority( getPriority() +1 );
     try {sleep(1000);}
     catch (InterruptedException ie) return;
   }
}
class ImplOfRunnable implements Runnable {
   public void run() { ... }
}
```

DEFINIÇÃO DE `run()` DE `Runnable`

- Somente pode-se chamar métodos de `Thread` se existir um objeto do tipo `Thread`:

```
class ImplOfRunnable implements Runnable {
   public void run() {
     System.out.println("Implementation of Runnable running");
 ??? setPriority( getPriority() +1 );
 ??? try {sleep(1000);}
     catch (InterruptedException ie) return;
   }
}
```

- A solução é criar objeto do tipo `Thread`:

```
class ImplOfRunnable implements Runnable {
   public void run() {
     Thread t = Thread.currentThread();
     System.out.println("Implementation of Runnable running");
     t.setPriority(t.getPriority() +1 );
     try {t.sleep(1000);}
     catch (InterruptedException ie) return;
   }
}
```

13.2 Classe Thread

A CLASSE `java.lang.Thread`

A classe possui os seguintes métodos:

- `Thread()`:
  Constrói uma nova linha, cujo fluxo de execução é dado pelo método `run`
- `Thread(Runnable r)`:
  Constrói uma nova linha, cujo fluxo de execução é dado pelo `run` do objeto r.
- `public void start()`:
  Inicia a execução da linha e dispara o método `run()`
- `public void run()`:
  Função principal que define o corpo da linha. Não deve ser chamada diretamente pelo usuário. Nada faz. Existe para ser redefinida por subclasses.
- `public final void stop()`:
  Encerra a execução da linha (depreciado).
- `public static void yield()`:
  Faz a linha ceder a vez. Se houver outras linhas executáveis, elas serão escaladas em seguida.
  O escalador escolhe a linha executável de maior prioridade, possívelmente a linha que cedeu a vez.
- `public static void sleep(long ms) throws InterruptException`:
  coloca a linha atualmente em execução para *dormir* por `ms` milisegundos, e permite que outras linhas tenham a chance de ser executadas.
- `public void interrupt()`:
  - causa o levantamento da exceção `InterruptException` na linha do objeto receptor, o qual deve estar no estado dormente ou de espera.
  - Se uma linha `t1` executa `t2.interrupt()`, quem recebe a exceção levantada é `t2` e não `t1`.
  - linha `t2` acima somente recebe a interrupção se estiver bloqueada
- `public static boolean interrupted()`:
  informa se a linha que executa a instrução recebeu algum pedido de interrupção e faz o status de interruped igual a false.
- `public final void join() throws InterruptedException`:
  Espera até que a linha (**this**) encerre sua execução.

- `public final  void join(long millis)`
  `throws InterruptedException`:
  Espera até que a linha (**this**) encerre sua execução no máximo o tempo `millis` (milissegundos) especificado.
  O tempo `0` significa espera indefinida.
- `public final`
  `void join(long millis, int nanos)`
  `throws InterruptedException`:
  Espera até que a linha (**this**) encerre sua execução no máximo o tempo de `millis*1000 + nanos` (nanossegundos) especificado.
  O tempo `0`(zero) nanossegundos significa espera indefinida.
- `public final void suspend()`:
  Suspende a execução da linha (depreciado por perigo de deadlock).
- `public final void resume()`:
  Retoma a execução da linha. Somente é válido depois que `suspend()` tiver sido chamado (depreciado).
- `public final boolean isAlive()`:
  Retorna verdadeiro se a linha tiver sido iniciada e ainda não parou.
  Retorna falso se a linha ainda for nova e não for executável, ou se tiver sido encerrada.
- `public final String getName()`:
  Retorna o string passado ao construtor de `Thread`.
- `public final setName(String name)`
  `throws securityException`
- `public final static int MIN_PRIORITY`:
  Prioridade mínima (1) que uma linha pode ter.
- `public final static int NORM_PRIORITY`:
  Prioridade default de uma linha (5).
- `public final static int MAX_PRIORITY`:
  Prioridade máxima (10) de uma linha
- `public static Thread currentThread()`:
  retorna referência da thread que executou o método;
- `public final void setPriority(int p)`
  `throws IllegalArgumentException`:
  Define prioridade da linha
  (`MIN_PRIORITY <= p <= MAX_PRIORITY`)
- `public final int getPriority()`:
  Informa a prioridade da linha.
- Há outros métodos.

13.3 Ciclo de Vida de *Threads*

CICLO DE VIDA DE UMA LINHA

- Linha x do tipo T é criada com a criação do objeto
  `T x = new T();`
- Linha x iniciada com a chamada

```
x.start();
```

- Execução inicia-se pelo método `x.run()`
- Linha x termina com o fim de `x.run()` ou com a chamada `x.stop()` – depreciado.
- Linha x pode ser interrompida temporariamente via `x.suspend()` – depreciado
- Linha x pode ser retomada via `x.resume()` – depreciado.
- Linhas estão sujeitas à **preempção** por uma outra linha de maior prioridade
- Uma linha pode ceder o controle do processador via `this.yield()`
- Prioridades vão de 1 (baixa) a 10 (alta)
- Pode-se ajustar a prioridade da linha x, como em
  `x.setPriority(x.getPriority() + 1)`
- Cada linha possui uma pilha tamanho padrão de 400kb aproximadamente

ESTADOS DE UMA LINHA

- NOVO:
  Estado inicial após a criação da linha com o operador `new`.
- EXECUTÁVEL:
  Estado após a execução do método `start()`.
  - 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 sistemas, usa-se a política de tempo compartilhado.
- BLOQUEADO:
  Uma linha entra no estado bloqueado quando é:
  - chamado o método `sleep()` da linha
  - chamado o método `suspend()` da linha (depreciado)
  - chamado o método `wait()` da linha
  - chamada uma operação de entrada/saída
- ENCERRADO:
  Uma linha é encerrada quando:
  - seu método `run()` terminou
  - o método `stop()` da linha foi executado (depreciado)

SAINDO DE ESTADO BLOQUEADO

- Quando uma linha bloqueada é reativada, o escalonador somente lhe dá o controle se ela tiver prioridade mais alta do que a atualmente em execução.
- A passagem estado BLOQUEADO para EXECUTÁVEL ocorre:
  - se a linha que *dormia* completou o tempo `ms` do `sleep(ms)`
  - se a linha suspensa teve seu método `resume()` chamado por uma outra.
  - se linha estava bloqueada devido a um `wait(..)` e `notify` ou `notifyAll` foi chamado

  - se a operação de E/S que a linha requisitou terminou.

- A ativação de uma linha somente é efetiva se combinar com a razão que a bloqueou. Por exemplo:
  - Se uma linha está bloqueada por I/O, uma chamada ao seu método `resume()` não muda seu estado.
- A ativação de um método incompatível com o estado da linha levanta a exceção `IllegalThreadStateException`.

13.4 Prioridades de *Threads*

PRIORIDADES DE LINHAS

- Toda linha tem uma prioridade.
- Toda linha herda a prioridade de seu criador.
- Pode-se aumentar ou diminuir a prioridade de uma linha `x` com chamada a `x.setPriority(p)`.
- O escalador de linhas sempre escolhe a linha de maior prioridade.
- O sistema de execução Java pode suspender uma linha de maior prioridade para que outras tenham a chance, mas não se tem garantia de quando isto ocorrerá.
- A linha executável de maior prioridade continua sendo executada até que ela:
  - ceda a vez via `yield()`
  - deixe de ser executável
  - seja substituída por linha de maior prioridade que se torne executável (acordou ou teve I/O completado ou alguém chamou `resume` ou `notify`)
  - sua fatia de tempo venceu (em algumas implementações)
- Escalador usa alguma política que não faz parte da linguagem Java quando houver mais de uma linha executável com a mesma prioridade.

USO DE `yield`

- Aplicação recebe uma lista de palavras e cria uma linha para cada palavra.
- O primeiro parâmetro da linha de comando indica se cada linha deve ou nao **ceder** a vez após o comando de impressão.
- O segundo parâmetro é o número de vezes a palavra será impressa.
- Demais parâmetros são a lista de palavras.

POR EXEMPLO:

```
>java Babel false 3 Sim Nao
>Sim Sim Sim Nao Nao Nao
>java Babel true 3 Sim Nao
>Sim Nao Sim Nao Sim Nao
```

```
class Babel extends Thread {
   static boolean deveCeder;
   static int vezes;
   String palavra;
   Babel(String oQue) { palavra = oQue; }
   public void run() {
      for (int i = 0; i < vezes; i++) {
          System.out.print(palavra + " ");
          if (deveCeder) yield();
      }
   }
   public static void main(String[] args) { ... }
}


 public static void main(String[] args) {
     vezes = Integer.parseInt(args[1]);
     deveCeder = new Boolean(args[0]).booleanValue();

     Thread corrente = currentThread();
     corrente.setPriority(Thread.MAX_PRIORITY);
     for (int i=2; i<args.length; i++) {
          new Babel(args[i]).start();
     }
 }
>java Babel false 3 Sim Nao
>Sim Sim Sim Nao Nao Nao
>java Babel true 3 Sim Nao
>Sim Nao Sim Nao Sim Nao
```

13.5 Sincronização de *Threads*

LINHAS RELACIONADAS (S/ SINCRONISMO)

- Determina se número `n` é primo.
- Valor `n` é o primeiro parâmetro da linha de comando.
- São criadas várias linhas do tipo `TestRange`, que procuram, em paralelo, divisores de `n` em centenas disjuntas.
- Cada linha lista os divisores encontrados.

```
class TestRange extends Thread { ... }

public class TestPrime {
  public static void main(String s[]) {
    long n = Long.parseLong(s[0]);
```

```
    int centuries = (int)(n/100) + 1;

    for(int i=0; i<centuries; i++) {
       new TestRange(n,i*100).start();
       System.out.println("starting ...");
    }
  }
}

class TestRange extends Thread {
   static long n; // numero a testar
   long from, to; // limites do intervalo

   TestRange(long n, int argFrom) {
       this.n = n;
       if (argFrom==0) {
           from=2;
       } else from=argFrom;
       to=argFrom+99;
   }
   public void run() {...}
}

public void run() {
   for (long i=from; i<=to && i<n; i++) {
        if (n%i == 0) {
           System.out.println("Factor " + i +
                    " found by thread " + getName());
           break;
        }
        yield();
   }
}
```

LINHAS MUTUAMENTE EXCLUSIVAS

- Manômetro controlado por várias linhas
- Cada linha deve testar se a pressão é inferior a máximo permitido antes de incrementá-la.
- Programa principal imprime o valor final da pressão quando todas as linhas houverem terminado.

```
public class Pressure extends Thread { ... }

public class Controle { ... }

public class Controle {
   public static int pressureGauge = 0;
```

```
   public static final int INCR    = 30;
   public static final int Limit   = 200;
   public static void main(String[]args) {
      Pressure []p = new Pressure[10];
      for (int i=0; i<10; i++)  {
          p[i] = new Pressure(); p[i].start();
      }
      try{ for(int i=0;i<10;i++) p[i].join();}
      catch(Exception e){ }
      System.out.println("gauge reads " +
              pressureGauge + ", safe limit is " + Limit);
}
```

CLASSE `Pressure`: PROPOSTA I(errada)

```
public class Pressure extends Thread {
  void RaisePressure() {
    if (Controle.pressureGauge < Controle.Limit-Controle.INCR){
        try {sleep(1000);} catch (Exception e){};
        Controle.pressureGauge += Controle.INCR;
    };
  }

  public void run() {RaisePressure(); }
}
```

- A solução acima não funciona devido a **condição de corrida**
- O valor final de `Controle.pressureGauge` é imprevisível

CLASSE `Pressure`: PROPOSTA II

```
class Pressure extends Thread {
  static synchronized void RaisePressure() {
    if (Controle.pressureGauge<Controle.Limit-Controle.INCR){
        try {sleep(1000);} catch (Exception e){};
        Controle.pressureGauge += Controle.INCR;
    };
  }

  public void run() {RaisePressure(); }
}
```

- Somente uma linha de cada vez pode executar o método `RaisePressure`, porque ele é **sincronizado**.
- `sleep(t)` suspende a execução por `t` nanosegundos, mas não libera o **bloqueio**.

CLASSE `Pressure`: PROPOSTA III

```
class Pressure extends Thread {
  static Object sync = new Object();
  void RaisePressure() {
   synchronized(sync) {
     if (Controle.pressureGauge<Controle.Limit-Controle.INCR){
        try {sleep(1000);} catch (Exception e){};
        Controle.pressureGauge += Controle.INCR;
     };
   }
  }

  public void run() { RaisePressure(); }
}
```

COMANDO `synchronized`

- O comando
  `synchronized(obj) {cmds}`
  bloqueia o objeto `obj` durante a execução de `cmds`.
- Outras linhas que executarem o comando `synchronized (obj)` para o mesmo objeto ficarão em espera até que o objeto seja desbloqueado.

```
class L1 extends Thread {
  Object lock;
  public L1(Object obj){lock = obj; }
  ...
  public void run(){... synchronized(lock){...r1...}; ...}
}
class L2 extends Thread {
  Object lock;
  public L1(Object obj){lock = obj;}
  ...
  public void run(){... synchronized(lock){...r2...};...}
}

class M {
  static public void main(String[] args) {
    A x = new A(); A y = new A();
    L1 t1 = new L1(x); L1 t3 = new L1(y);
    L2 t2 = new L2(x); L2 t4 = new L2(y);
    t1.start(); t2.start(); t3.start(); t4.start();
}
```

CLASSE `Pressure`: PROPOSTA IV (errada)

```
class Pressure extends Thread {
```

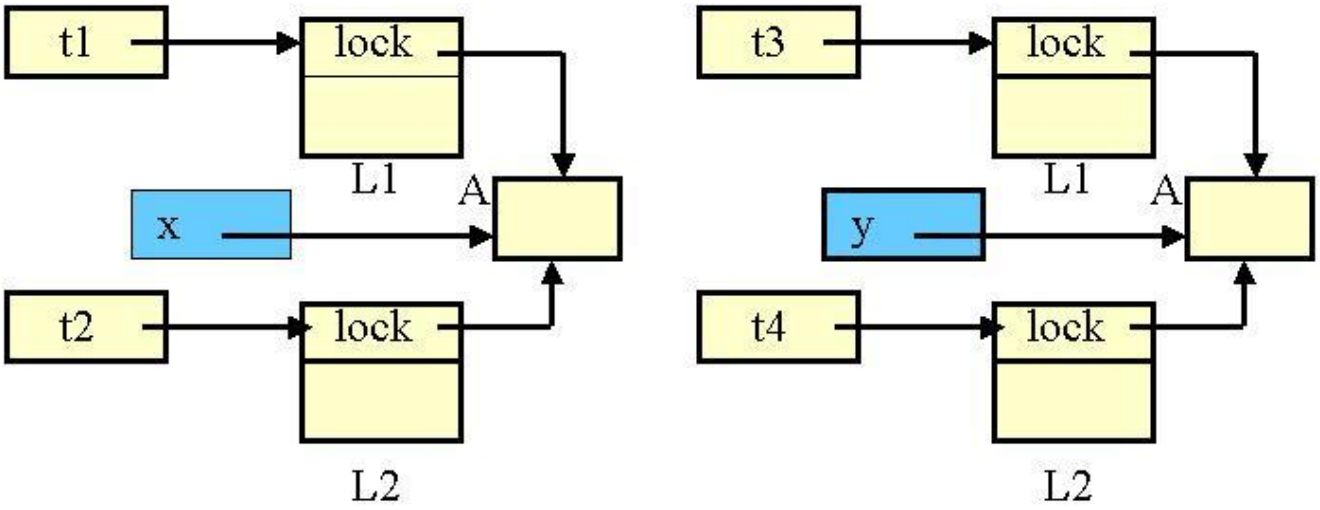


Figura 13.1

```
  synchronized void RaisePressure() {
     if (Controle.pressureGauge < Controle.Limit - Controle.INCR) {
      try {sleep(100);}
          catch (Exception e){};
      Controle.pressureGauge += 15;
     };
  }
  public void run() {RaisePressure(); }
}
```

- Não funciona, porque

  ```
  synchronized void RaisePressure() {
      corpo
  }
  ```

  é equivalente a

  ```
  void RaisePressure() {
    synchronized(this) { corpo }
  }
  ```

- `synchronized` somente sincroniza métodos de um **mesmo** objeto.

MÉTODOS SINCRONIZADOS

- Métodos `sincronizados` de um mesmo objeto são mutuamente exclusivos
- Métodos `sincronizados` NÃO excluem métodos não sincronizados ou métodos estáticos (sincronizados ou não).
- Um método sincronizado pode chamar sem bloqueio qualquer outro também sincronizado com mesmo `this`.
- A redefinição de um método sincronizado pode ou não ser `synchronized`.
- `super.method()` pode ser sincronizado mesmo quando `method()` não for.

- Se um método não-sincronizado faz uma chamada ao seu `super` sincronizado, durante a execução do `super()` o objeto fica bloqueado.

OPERAÇÕES DE SINCRONIZAÇÃO

- Métodos `wait` e `notify` estão definidos na classe `Object`
- O método `wait` permite a espera por uma condição
- O método `notify` permitir informar àqueles em espera que alguma coisa aconteceu.
- `wait` e `notify` se aplicam a objetos particulares, da mesma forma que bloqueios, isto é, `x.notify()` notifica uma linha que executou `y.wait()`, se `x` e `y` denotarem o mesmo objeto.
- `wait` e `notify` se aplicam somente a objetos monitores, isto é, objetos que tem métodos ou comandos synchronized.
- A execução do `x.wait()` libera bloqueio do objeto.
- Quando a linha é reativada, o bloqueio do objeto é restabelecido.
- Todo objeto `x` do tipo monitor têm duas filas associadas:
  - fila de bloqueio de *threads* a espera da liberação do objeto `x`
  - fila de espera de *threads* que executaram `x.wait()`
- `x.notify()` pode ser ativado de qualquer lugar.
- `x.notify()` acorda apenas uma linha, isto é, passa uma das linhas da fila de espera de `x` para a respectiva fila de bloqueio de sincronização do objeto.
- Para acordar todas as linhas na fila de espera de um objeto `x` usa-se `x.notifyAll()` em vez de `x.notify()`.

ESTRUTURA DE PROGRAMA COM SINCRONIZAÇÃO

- Linha que espera a `condição`:

```
synchronized void execQuandoPuder(){
  ...
  while(!condicao) { wait(); };
  ... neste ponto condicao e' verdadeira
}
```

- Linha que cria a `condição`:

```
synchronized void FazPoder() {
   ... muda algum valor usado na
       condicao do execQuandoPuder();
       notifyAll(); ....
}
```

DETALHES DE `wait`

- `public final void wait(long timeout)`
  `throws InterruptedException`:
  A linha corrente espera ser notificada, via o objeto corrente, mas espera no

máximo o tempo especificado `timeout` (ms).
Se `timeout==0` espera notificação indefinidamente.

- `public final void wait() throws InterruptedException`:
  O mesmo que `wait(0)`.
- `public final void wait(long mili, int nanos) throws InterruptedException`:
  O tempo de espera é a soma de milissegundos com nanossegundos.

DETALHES DE `notify`

- `public final void notify()`:
  Notifica exatamente **uma** das linhas que esperam por uma condição via um `wait` no objeto corrente do `notify`.
  Não é possível escolher quem será notificado. Use com cuidado.
- `public final void notifyAll()`:
  Notifica **todas** as linhas que esperam por alguma condição.

13.6 *Threads* Sincronizadas

PROBLEMA DO PRODUTOR X CONSUMIDOR

- Problema do Consumidor e Produtor que se comunicam via um buffer.
- Um produtor continuamente deposita, um a um, inteiros em um buffer
- Um consumidor retira continuamente um inteiro do buffer.

```
class Buffer { ... }

class Consumer extends Thread { ... }

class Producer extends Thread { ... }

public class Main{
  public static void main(String args[ ]) {
      Buffer b = new Buffer(100);
      Consumer c1 = new Consumer(b), c2 = new Consumer(b);
      Producer p1 = new Producer(b), p2 = new Producer(b);
      c1.start( ); p1.start( );  c2.start( ); p2.start( );
  }
}
```

DEFINIÇÃO DO CONSUMIDOR E PRODUTOR

```
class Consumer extends Thread {
   private Buffer b;
   public Consumer(Buffer b) {this.b = b; }
   public void run( ) {
```

```
       int i;
       while (true) {
             i = b.get();
             System.out.print(i);
             yield( );
       }
   }
}

class Producer extends Thread {
    private Buffer b;
    public Producer(Buffer b) { this.b = b; }
    public void run( ) {
         int i = 0;
         while (true) {
               b.put(i++);
               yield( );
         }
    }
}
```

DEFINIÇÃO DO BUFFER

```
public class Buffer {
   private int size;
   private int [ ] contents;
   private int in, out;
   private int total =0;
   Buffer(int size) {
      this.size = size;  contents = new int[size];
   }
   public synchronized void put(int item){ ...}
   public synchronized int get() {... }
}

public synchronized void put(int item){
    while(total >= size) {
        try {this.wait();} catch(InterruptedException e) { };
    }
    contents[in] = item;
    System.out.println("Buffer:write at "+ in + "item" + item);
    if(++in == size) in = 0;
    total++;
    this.notifyAll();
}

public synchronized int get() {
    int temp;
```

```
    while(total <= 0)
        try {this.wait();} catch(InterruptedException e) { };
    temp = contents[out];
    System.out.println("Buffer:read from "+out+ "item" + temp);
    if(++out == size) out=0;
    total--;
    this.notifyAll();
    return temp;
}
```

LINHAS EGOÍSTAS E COOPERATIVAS

- Em alguns sistemas, linha monopiliza o controle até seu término.
- Em outros, como o Windows NT, o sistema implementa a 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 qual o ambiente o programa com as linhas será executado.
- RECOMENDA-SE que toda linha chame `yield` ou `sleep` quando estiver executando um loop long, assegurando-se que não estará monopolizando o sistema.

13.7 Encerramento de *Threads*

- Toda aplicação começa com uma linha que executa seu método `main`.
- Uma aplicação que não cria outras linhas termina quando seu `main` termina.
- Linhas podem ser user ou daemon.
- Por default, toda linha é do tipo user.
- No caso de haver outras linhas, a aplicação somente termina quando todas as linhas do tipo user terminarem.
- Aplicação não espera por linhas do tipo *daemon*, isto é, todas linhas deste tipo morrem junto com a aplicação.
- Criam-se linhas daemon com o comando:
  `setDaemon(true)`

FIM DE UMA LINHA

- Uma linha `t` termina quando:
  - seu `run` termina
  - `t.stop()` é executado como último recurso (depreciado).
- Comando `t.stop()` lança a exceção `ThreadDeath`, uma extensão de `Error`.
- Se `ThreadDeath` não for capturada, a linha `t` é encerrada sem emitir qualquer mensagem.
- A exceção levantada por `this.stop()` pode ser capturada pela própria `Thread this`, anulando sua *morte*.
- Uma linha pode executar diretamente o comando
  `throw new ThreadDeath()`.

FIM DE UMA LINHA (EXEMPLO I)

```
import java.io.*;
class T extends Thread {
  public void run() {
    PrintStream o = System.out;
    o.println("T1. Begin");
    stop();
    o.println("T2. Resurrection");
  }
}
SAIDA: M1. Begin
       T1. Begin
       M3. End

public class Stop1{
  static public void main(String[] args) {
    PrintStream o = System.out;
    T t = new T(); o.println("M1. Begin"); t.start();
    try {Thread.sleep(2000);}
        catch(InterruptedException e)
                    {o.println("M2. Interrupted");};
    o.println("M3. End");
  }
}
SAIDA: M1. Begin
       T1. Begin
       M3. End
```

FIM DE UMA LINHA (EXEMPLO II)

```
import java.io.*;
class T extends Thread {
  public void run() {
    PrintStream o = System.out;
    o.println("T1. Begin");

    try {stop();}
    catch(ThreadDeath d){o.println("T2. Post-Mortem Sigh ");}

    o.println("T3. Resurrection");
  }
}

public class Stop2{
  static public void main(String[] args) {
    PrintStream o = System.out;
    T t = new T();
```

```
    System.out.println("M1. Begin");
    t.start();
    System.out.println("M2. Continues");
    try{ t.join();}
    catch(ThreadDeath d){o.println("M3. Faces Death");}
    catch(InterruptedException d){o.println("M4. Interrupted");}
   o.println("M5. End ");
  }
}

SAIDA: M1. Begin
       M2. Continues
       T1. Begin
       T2. Post-Mortem Sigh
       T3. Resurrection
       M5. End
```

FIM DE UMA LINHA (EXEMPLO III)

```
import java.io.*;
class T extends Thread {
  public void run() {
    PrintStream o = System.out;
    o.println("T1. Begin");
    try{sleep(3000);}
       catch(InterruptedException d){
         o.println("T2. Interrupted");}
    o.println("T3. End");
  }
}

public class Stop3{
  static public void main(String[] args) {
    PrintStream o = System.out;
    T t = new T(); o.println("M1. Begin");
    t.start(); o.println("M2. Continues");
    try {Thread.sleep(1000);}
    catch(InterruptedException d){o.println("M3. Interrupted");}
    o.println("M4. Continues");
    try{ t.stop();}
    catch(ThreadDeath d){o.println("M5. Faces Death");}
    o.println("M6. End");
  }
}

SAIDA:
M1. Begin
M2. Continues
```

```
T1. Begin
M4. Continues
M6. End
```

FIM DE UMA LINHA (EXEMPLO IV)

```
import java.io.*;
class T extends Thread {
  public void run() {
    PrintStream o = System.out;
    o.println("T1. Begin");
    try{sleep(3000);}
       catch(InterruptedException d){
         o.println("T2. Interrupted");}
       catch(ThreadDeath d){
         o.println("T3. Faces Death");}
    o.println("T4. End");
  }
}

public class Stop4{
  static public void main(String[] args) {
    PrintStream o = System.out;
    T t = new T(); o.println("M1. Begin"); t.start();
    o.println("M2. Continues");
    try {Thread.sleep(1000);}
    catch(InterruptedException d){o.println("M3. Interrupted");}
    o.println("M4. Continues");
    try{ t.stop();}
    catch(ThreadDeath d){o.println("M5. Faces Death");}
    o.println("M7. End");
  }
}

SAIDA DE Stop4: M1. Begin
                M2. Continues
                T1. Begin
                M4. Continues
                M7. End
                T3. Faces Death
                T4. End
```

FIM DE UMA LINHA (EXEMPLO V)

```
import java.io.*;
class T extends Thread {
  public void run() {
    PrintStream o = System.out;
```

```
    o.println("T1. Begin");
    try{sleep(3000);}
       catch(InterruptedException d){
         o.println("T2. Interrupted");}
       catch(ThreadDeath d){
         o.println("T3. Faces Death");}
    o.println("T4. End");
  }
}

public class Stop5{
  static public void main(String[] args) {
    PrintStream o = System.out;
    T t = new T(); o.println("M1. Begin"); t.start();
    o.println("M2. Continues");
    try {Thread.sleep(1000);}
    catch(InterruptedException d){
         o.println("M3. Interrupted");}
    o.println("M4. Continues");
    t.interrupt();
    o.println("M5. End");
  }
}

SAIDA DE Stop5:
M1. Begin
M2. Continues
T1. Begin
M4. Continues
M5. End
T2. Interrupted
T4. End
```

13.8 Exercícios

13.9 Notas Bibliográficas

Capítulo 14

Reflexão Computacional

Há situações em que o acesso durante a execução de informações sobre os dados ou o código do programa em execução se faz necessário ou conveniente. Em outros casos, a alteração de objetos. Esta atividade de se focalizar nos elementos intrínsecos de um programa, solicitar-lhe informações de si e manipulá-los é chamada de **reflexão computacional**.

Java oferece dois recursos para se obter reflexão computacional: o método `getclass` da classe `Object` e a classe `Class`.

14.1 Classe Class

CLASSE `java.lang.Class`

- `String getName()`:
  `c.getname()` retorna o nome da class descrita por `c`
- `Class getSuperClass()`:
  `c.getSuperClass()` retorna a superclasse de `c`.
- `Class[ ] getInterfaces()`:
  `c.getInterfaces()` retorna um arranjo de objetos `Class` que informa as interfaces implementadas por `c`.
- `boolean isInterface()`
- `String toString()(`:
  `c.toString()` retorna o nome da classe ou interface descrita pelo objeto `c`.

CLASSE `java.lang.Class` **...**

- `static Class forName(String className)`:
  retorna o objeto `Class` que representa a classe de nome `className`
- `Object newInstance()`:
  retorna uma nova instância desta classe
- `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.

CLASSE `java.lang.Class` ...

- `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.
- `Constructor[ ] getDeclaredConstructors`:
  retorna um arranjo contendo os objetos `Constructor` que fornece todos os construtores;

OBTENÇÃO DO OBJETO `Class` DADO O OBJETO

- Uso de `getClass()`:

```
1          Empregado e;
2          Class c = e.getClass();
3          ...
4          Methods [ ] m = c.getMethods();
5
```

OBTENÇÃO DO OBJETO `Class` DADO O NOME DA CLASSE

- Uso de `forName(String)`:

```
1          String nome = "Empregado";
2          Class c = Class.forName(nome);
3          ...
4          Methods [ ] m = c.getMethods();
5          Object x = c.newInstance();
6
```

OBTENÇÃO DO NOME DE UMA CLASSE EXISTENTE

- O trecho de programa

```
1 Empregado e = new Empregado("José da Silva",data);
2 Class c = e.getclass();
3 System.out.println(c.getName() + " " + e.getname());
```

- IMPRIME `Empregado José da Silva`

CLASSES `java.lang.reflect.Field`, `Method` e `Constructor`

- `Class getDeclaringClass()`: retorna o objeto `Class` da classe que define este construtor, método ou campo.
- `Class[ ] getExceptionTypes()`: retorna um arranjo de objetos `class` que representam os tipos de exceções levantadas pelo método.
- `int getModifiers()`: retorna um inteiro que descreve os modificadores desse construtor, método ou campo.
  (Use classe `Modifier` para analisar o inteiro retornado).

- `String getName()`: retorna `String` que é o nome do construtor, método ou campo.
- `Class[ ] getParameterTypes`: retorna um arranjo de objetos `Class` representam os tipos dos parâmetros desse construtor ou método.

CLASSE `java.lang.reflect.Modifier`

- public static java.lang.String toString(int);
- public static boolean isInterface(int);
- public static boolean isPrivate(int);
- public static boolean isTransient(int);
- public static boolean isStatic(int);
- public static boolean isPublic(int);
- public static boolean isProtected(int);
- public static boolean isFinal(int);
- public static boolean isSynchronized(int);
- public static boolean isVolatile(int);
- public static boolean isNative(int);
- public static boolean isAbstract(int);
- public static boolean isStrict(int);

TESTE DA REFLEXÃO

```
1 import java.lang.reflect.*;
2 import CoreJava.*;
3 public class ReflectionTest {
4    public static void printConstructors(Class cl) {...}
5    public static void printMethods(Class cl){ ... }
6    public static void printFields(Class cl){ ... }
7    public static void main(String[] args) { ...}
8 }
```

TESTE DA REFLEXÃO ...

```
1  public static void main(String[] args) {
2     String name = Console.readLine
3         ("Please enter a class name (e.g. java.util.Date): ");
4     try {
5         Class cl = Class.forName(name);
6         Class supercl = cl.getSuperclass();
7         System.out.print("class " + name);
8         if (supercl != null && !supercl.equals(Object.class))
9             System.out.print(" extends " + supercl.getName());
10        //Imprime construtores, métodos e campos de cl
11        System.out.print("\n{\n");
12        printConstructors(cl);
13        System.out.println();
14        printMethods(cl);
15        System.out.println();
```

```
16          printFields(cl);
17          System.out.println("\n}");
18      }
19      catch(ClassNotFoundException e) {
20           System.out.println("Class not found.");
21      }
22  }
```

TESTE DA REFLEXÃO ...

```
1  public static void printConstructors(Class cl) {
2     Constructor[] constructors = cl.getDeclaredConstructors();
3     for (int i = 0; i < constructors.length; i++) {
4        Constructor c = constructors[i];
5        Class[] paramTypes = c.getParameterTypes();
6        String name = c.getName();
7        System.out.print("   " + Modifier.toString(c.getModifiers()));
8        System.out.print(" " + name + "(");
9        for (int j = 0; j < paramTypes.length; j++) {
10           if (j > 0) System.out.print(", ");
11           System.out.print(paramTypes[j].getName());
12       }
13       System.out.println(");");
14    }
15 }
```

TESTE DA REFLEXÃO ...

```
1  public static void printMethods(Class cl) {
2     Method[] methods = cl.getDeclaredMethods();
3     for (int i = 0; i < methods.length; i++) {
4       Method m = methods[i]; String name = m.getName();
5       Class retType = m.getReturnType();
6       Class[] paramTypes = m.getParameterTypes();
7       System.out.print("   " + Modifier.toString(m.getModifiers()));
8       System.out.print(" " +retType.getName()+ " " +name+ "(");
9       for (int j = 0; j < paramTypes.length; j++) {
10           if (j > 0) System.out.print(", ");
11           System.out.print(paramTypes[j].getName());
12      }
13      System.out.println(");");
14    }
15 }
```

TESTE DA REFLEXÃO ...

```
1  public static void printFields(Class cl) {
2     Field[] fields = cl.getDeclaredFields();
3     for (int i = 0; i < fields.length; i++) {
4        Field f = fields[i];
```

```
5         Class type = f.getType();
6         String name = f.getName();
7         System.out.print("   " + Modifier.toString(f.getModifiers()));
8         System.out.println(" " + type.getName() + " " + name
9            + ";");
10     }
11 }
```

SAÍDA DE `ReflectionTest` **PARA** `java.lang.Object`

```
1  class java.lang.Object {
2     public java.lang.Object();
3     public native int hashCode();
4     protected void finalize();
5     public final void wait();
6     public final void wait(long, int);
7     public final native void wait(long);
8     private static native void registerNatives();
9     public final native java.lang.Class getClass();
10    public boolean equals(java.lang.Object);
11    protected native java.lang.Object clone();
12    public java.lang.String toString();
13    public final native void notify();
14    public final native void notifyAll();
15 }
```

14.2 Exercícios

14.3 Notas Bibliográficas

Capítulo 15

Applets

15.1 Introdução

15.2 Criação de Applets

APPLETS

- Applets são um meio de executar um código Java a partir de um navegador (*browser*).
- Applets são definidas por um protocolo que governa seu tempo-de-vida e os métodos por meio dos quais pode-se inspecionar ou manipular o ambiente de execução.
- A superclasse `java.applet.Applet` define o ambiente.
- Em Java 2, com `swing`, deve-se usar

```
import javax.swing.JApplet
```

CLASSE `Applet`

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet
```

CLASSE `JApplet`

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
```

```
                    |
                    +--java.applet.Applet
                          |
                          +--javax.swing.JApplet
```

EXECUÇÃO DE APPLETS

- Quando rótulo `APPLET` ou `OBJECT` for encontrado na página que estiver sendo carregada, o navegador:
  - Carrega a código (bytecode) da classe especificada.
  - Cria um objeto desta classe.
  - Cria uma região na página da WEB para ser controlada pela applet.
  - Chama em seqüência os métodos `init`, `start` e `paint`.
- Durante a navegação outros métodos de applets são automaticamente chamados.
- Outras classes poderão ser carregadas ao longo da execução da applet.

CRIAÇÃO DE APPLETS

- Para criar uma applet, deve-se:
  - Criar um arquivo com extensão `.java` para ser o tipo da applet.
  - Criar nesse arquivo a classe que define a applet como uma extensão de `java.applet.Applet` ou de `javax.swing.JApplet`
  - Mencionar o nome da applet na página HTML, junto com definição de parâmetros, área de trabalho, etc.

UMA APPLET MUITO SIMPLES I

- Arquivo Mensagem.java:

```
1 import java.awt.Graphics; import java.applet.Applet;
2 public class Mensagem extends Applet {
3    public void paint(Graphics g) {
4       g.drawString("Sim!", 15, 25);
5    }
6 }
```

- Arquivo Mensagem.java:

```
1 import java.awt.Graphics; import javax.swing;
2 public class Mensagem extends JApplet {
3    public void paint(Graphics g) {
4       g.drawString("Sim!", 15, 25);
5    }
6 }
```

- Arquivo Pagina1.html:

```
1 <title> Applet muito simples</title>
2 <applet code="Mensagem.class" width=200 height=100>
3 </applet>
```

- Arquivo Pagina2.html:

```
<title> Applet muito simples</title>
<object codetype=application/java" classid = "Mensagem.class"
        width=200 height=100>
</object>
```

`Z:\>appletviewer Pagina1.html`:

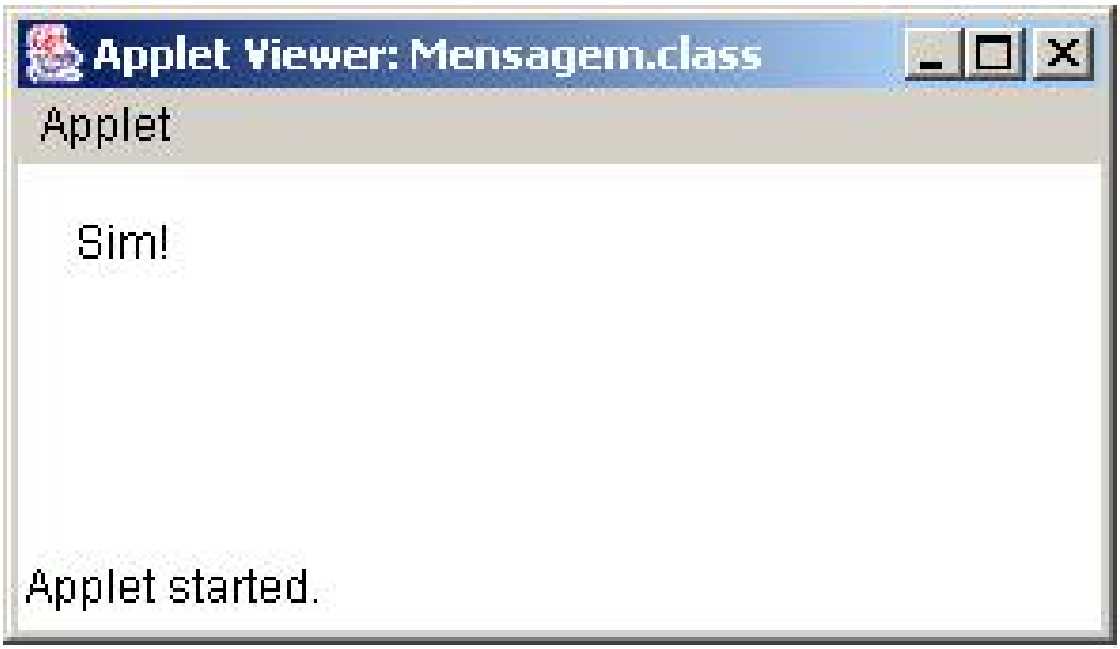


Figura 15.1 Primeiro Applet

UMA APPLET MUITO SIMPLES II

Arquivo `Teste.html`:

```
<HTML>
  <APPLET Code="Teste.class", width=300 height=40></APPLET>
</HTML>
```

Arquivo `Teste.java`:

```
import java.applet.Applet; import java.awt.Graphics;
public class Teste extends Applet {
   public void init() { repaint(); }
   public void paint(Graphics g) {
     g.drawRect(0, 0, size().width - 1, size().height - 1);
     g.drawString("Texto dentro do retangulo", 5, 15);
   }
}
```

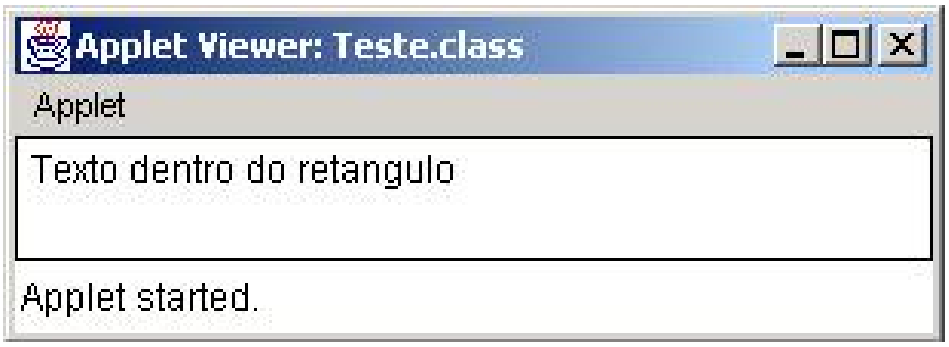


Figura 15.2 Segundo Applet

15.3 Ciclo de Vida de Applets

MÉTODOS DO CICLO DE VIDA DA APPLET

- `void init()`:
  Chamado automaticamente quando a applet for carregada a primeira vez.
  Se a applet usa threads, `init` deve tipicamente criar essas threads.
- `void start()`:
  Chamado automaticamente sempre que o browser visitar a página que contém a applet.
  Isto é, chamado em resposta à pressão dos botões **Back** e **Forward** do navegador.

- `void stop()`:
  Chamado automaticamente sempre que o navegador sair da página que contém a applet.
  Isto é, chamado em resposta à pressão dos botões **Back** e **Forward** do navegador.
- `void destroy()`:
  Chamado quando a página não for mais alcançável.
  É usado para liberar recursos não mais necessários.
- `void paint(Graphics g)`:
  chamado automaticamente pelo sistema de janela quando o componente tiver que ser (re)exibido.

FLUXO DE EXECUÇÃO DE APPLETS

1. Browser faz o primeiro acesso à página com applet embutida.
2. Applet referenciada é carregada.
3. Método `init()` é automaticamente chamado.
4. Browser chama o método `start()`.
   Applet precisa de `start` quando cria linhas de execução.
5. A seguir, o browser chama o método `paint()`, que é um método da janela para desenha a parte gráfica do applet.
6. Neste ponto a applet está sendo executada, isto é, seu efeito pode ser observado na página.

1. Quando o browser deixar a página, o método `stop()` é automaticamente chamado para fazer qualquer *limpeza* necessária.
2. Applet precisa de `stop()` sempre que *limpeza* for necessária.
3. Limpeza típica pode ser interromper linhas de execução geradas pela applet.
4. Todo retorno à página causa chamada automática a `start()` e a `paint()`.
5. Quando a página for descartada permanentemente, método `destroy()` é chamado.

15.4 Exemplos de Applets

APPLET SOMADOR(`AddtionApplet.html`)

```
<html>
<hr>
<h2>Applet ilustra entrada e saida de dados via browser</h2>

<applet code = "AdditionApplet.class" width = "300" height = "50">
</applet>

<h2>Area da applet está logo acima</h2>
<hr>
</html>
```

APPLET SOMADOR(`AddtionApplet.java`)

```
import java.awt.Graphics;
import javax.swing.*;
public class AdditionApplet extends JApplet {
   double sum;  // sum of values entered by user

   public void init() {  ... }

   public void paint( Graphics g ) { ... }
}
```

APPLET SOMADOR (init)

```
public void init() {
      String firstNumber;   // first string entered by user
      String secondNumber;  // second string entered by user
      double number1;       // first number to add
      double number2;       // second number to add
      firstNumber = JOptionPane.showInputDialog(
                  "Enter first floating-point value" );
      secondNumber = JOptionPane.showInputDialog(
                  "Enter second floating-point value" );
      number1 = Double.parseDouble( firstNumber );
      number2 = Double.parseDouble( secondNumber );
      sum = number1 + number2;
}
```

APPLET SOMADOR - LEITURA

Resultado após execução do método `init`:

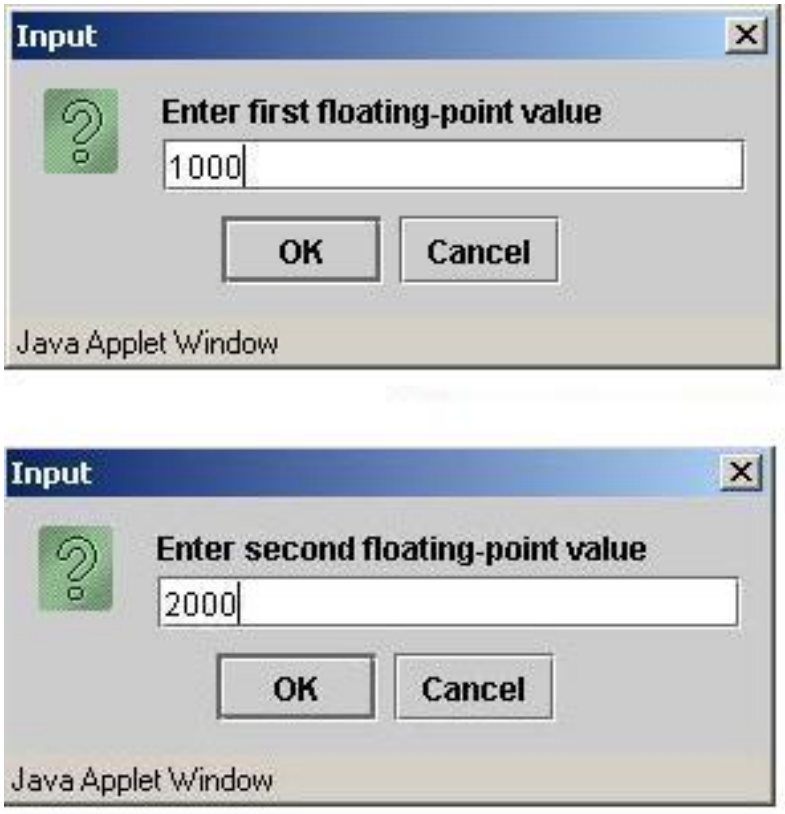


Figura 15.3 Applet Somador - Entrada

APPLET SOMADOR (paint)

```
1 // draw results in a rectangle on applet's background
2 public void paint( Graphics g ) {
3   super.paint( g );
4   // draw rectangle starting from (15, 10) that is 270
5   // pixels wide and 20 pixels tall
6   g.drawRect( 15, 10, 270, 20 );
7   // draw results as a String at (25, 25)
8   g.drawString( "The sum is " + sum, 25, 25 );
9 }
```

APPLET SOMADOR - RESULTADOS

Após execução do método `paint`:

APPLET SOMADOR(`AddtionApplet.java`) Versão II

```
1 import java.awt.Container;
```

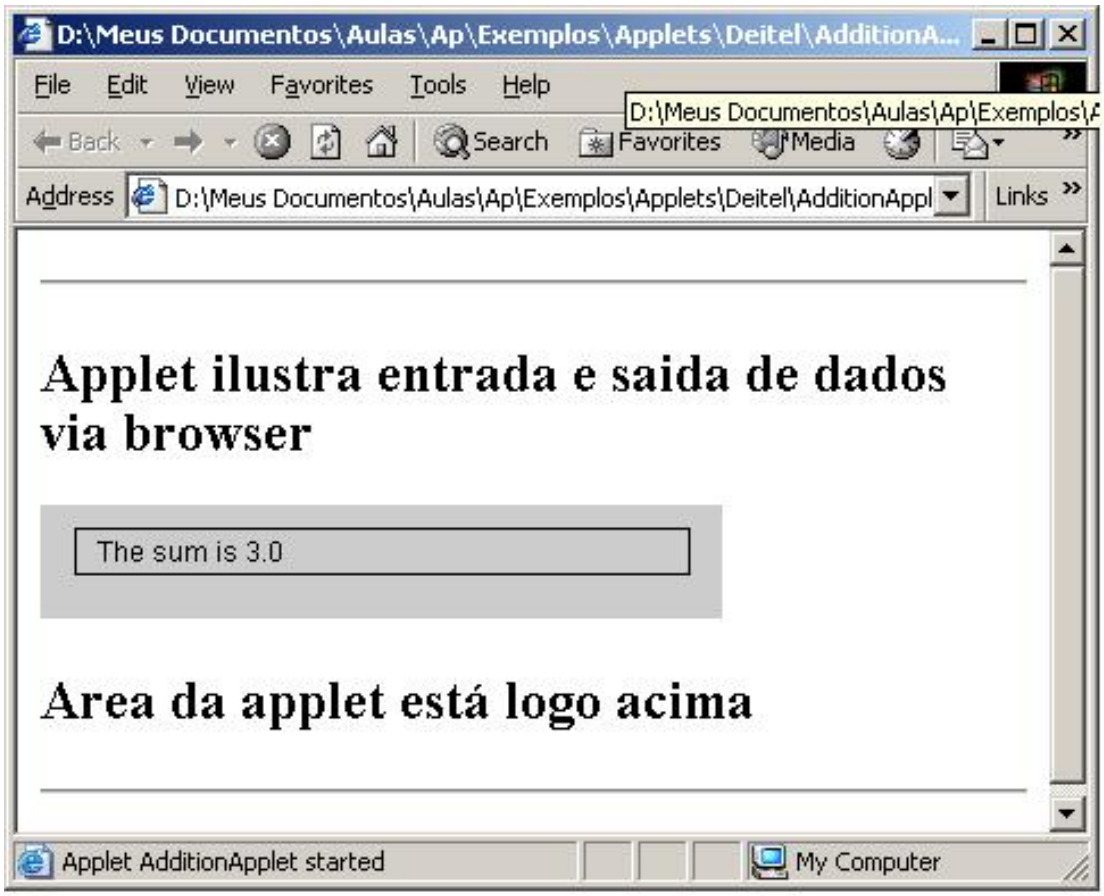


Figura 15.4 Applet Somador - Resultado

```
2  import javax.swing.*;
3  public class AdditionApplet extends JApplet {
4     double sum;
5     public void init() {
6        String firstNumber;  String secondNumber;
7        double number1; double number2;
8        JTextArea outputArea = new JTextArea();
9        Container container = getContentPane();
10       container.add(outputArea);
11       firstNumber = JOptionPane.showInputDialog(
12          "Enter first floating-point value" );
13       secondNumber = JOptionPane.showInputDialog(
14          "Enter second floating-point value" );
15       number1 = Double.parseDouble( firstNumber );
16       number2 = Double.parseDouble( secondNumber );
17       sum = number1 + number2;
18       outputArea.setText("The sum is " + sum);
19    }
```

```
20 }
```

TRANSFERINDO PARÂMETROS

- Obedece a mesma convenção de argumentos na linha de comando de main, isto é, os parametros são transferidos como `Strings`.
- Função `getParameter(nome do param)` retorna o string que representa o valor.
- No arquivo anagrama.html:

```
1 ...
2 <applet code="Anagrama.class" width=500 height=500>
3    <param name="objetivo" value="surfando na rede">
4    <param name="palavras" value="palavra.txt">
5    <param name="tamanho" value="2">
6    <param name="valor" value="3.14">
7 </applet>
8 ...
```

- No arquivo Anagrama.java:

```
1 String s1 = getParameter("tamanho");
2 int k = Integer.parseInt(s1);
3 String s2 = getParameter("objetivo");
4 String s3 = getParameter("valor");
5 double d  = Double.valueOf(s3).doubleValue();
```

CAlCULADORA - SEM APPLET

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 public class Calculator {
5    public static void main(String[] args) {
6        JFrame frame = new CalculatorFrame( );
7        frame.show();
8     }
```

SAÍDA DE `java Calculator`

QUADRO DA CALCULADORA

```
1 class CalculatorFrame extends JFrame {
2   public CalculatorFrame() {
3       setTitle("Calculator");
4       setSize(200, 200);
5       addWindowListener(new WindowAdapter( ) {
6           public void windowClosing(WindowEvent e){
7               System.exit(0);
8           }
9       } );
10
```

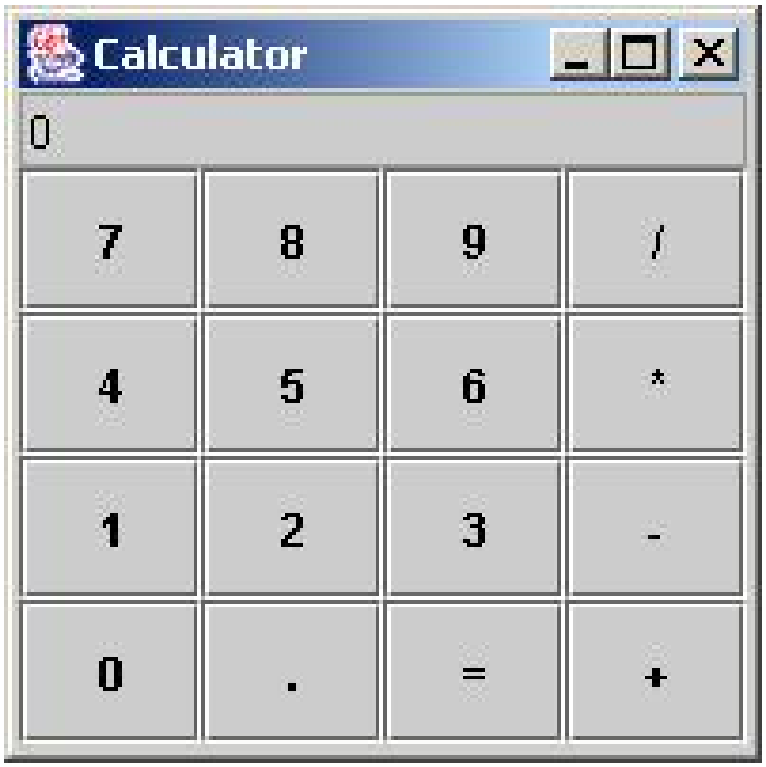


Figura 15.5 Interface da Calculadora

```
11        Container contentPane = getContentPane();
12        contentPane.add(new CalculatorPanel());
13    }
14 }
```

PAINEL DO QUADRO DA CALCULADORA

```
1  class CalculatorPanel extends JPanel implements ActionListener {
2     private JTextField display;
3     private double arg = 0;
4     private String op = "=";
5     private boolean start = true;
6     public CalculatorPanel() ...
7     private void addButton(Container c, String s){...}
8     public void actionPerformed(ActionEvent evt){...}
9     public void calculate(double n){...}
10 }
```

CONSTRUTOR DE `CalculatorPanel`

```
1  public CalculatorPanel() {
2        setLayout(new BorderLayout());
3        display = new JTextField("0");
4        display.setEditable(false);
```

```
5        add(display, "North");
6        JPanel p = new JPanel();
7        p.setLayout(new GridLayout(4, 4));
8        String buttons = "789/456*123-0.=+";
9        for (int i = 0; i < buttons.length(); i++)
10          addButton(p, buttons.substring(i, i + 1));
11       add(p, "Center");
12   }
```

MÉTODO addButton

```
1  private void addButton(Container c, String s) {
2     JButton b = new JButton(s);
3     c.add(b);
4     b.addActionListener(this);
5  }
```

MÉTODO actionPerformed

```
1  public void actionPerformed(ActionEvent evt) {
2     String s = evt.getActionCommand();
3     if ('0'<=s.charAt(0) && s.charAt(0)<='9'||s.equals(".")){
4        if (start) display.setText(s);
5        else display.setText(display.getText() + s);
6        start = false;
7     } else {
8        if (start) {
9           if (s.equals("-")){display.setText(s); start = false;}
10          else op = s;
11       } else {
12          double x = Double.parseDouble(display.getText());
13          calculate(x); op = s; start = true;
14       }
15    }
16 }
```

MÉTODO Calculate

```
1  public void calculate(double n) {
2        if (op.equals("+")) arg += n;
3        else if (op.equals("-")) arg -= n;
4        else if (op.equals("*")) arg *= n;
5        else if (op.equals("/")) arg /= n;
6        else if (op.equals("=")) arg = n;
7        display.setText("" + arg);
8  }
```

A PÁGINA DA CALCULADORA

```
1  <HTML>
2  <TITLE> Uma Calculadora </TITLE>
```

```
<BODY>
  Eis aqui uma calculadora.
<HR>

<APPLET code = "CalculatorApplet.class" width=250 height=200>
</APPLET>

<HR>

</BODY>
</HTML>
```

CAlCULADORA APPLET

- No lugar das classes `Calculator` e `CalculatorFrame` tem-se apenas `CalculatorApplet`
- Classe `CalculatorPanel` fica inalterada.
- Define-se o Applet com sendo:

```
public class CalculatorApplet extends JApplet {
   public void init( ) {

      Container contentPane = getContentPane();
      contentPane.add(new CalculatorPanel( ) );

   }
}
```

SAIDA DE `appletviewer CalculatorApplet.html`

SAIDA DO NAVEGADOR PARA `CalculatorApplet.html`

APPLETS X SEGURANÇA

- Applets são executadas sob controle *sandbox*
- No `sandbox` operações perigosas como acesso a arquivos, acesso à rede, execução de programa local, são proibidas ou limitadas pelo **security manager**.
- Classes podem ser assinadas digitalmente de forma a dar garantias quanto sua identificação.
- Applets confiáveis podem receber direitos de acessos privilegiados.
- Usuários podem configurar seus *browsers* com uma lista de organizações confiáveis, cujos applets podem ser executados automaticamente com mais privilégios.

15.5 Exercícios

15.6 Notas Bibliográficas

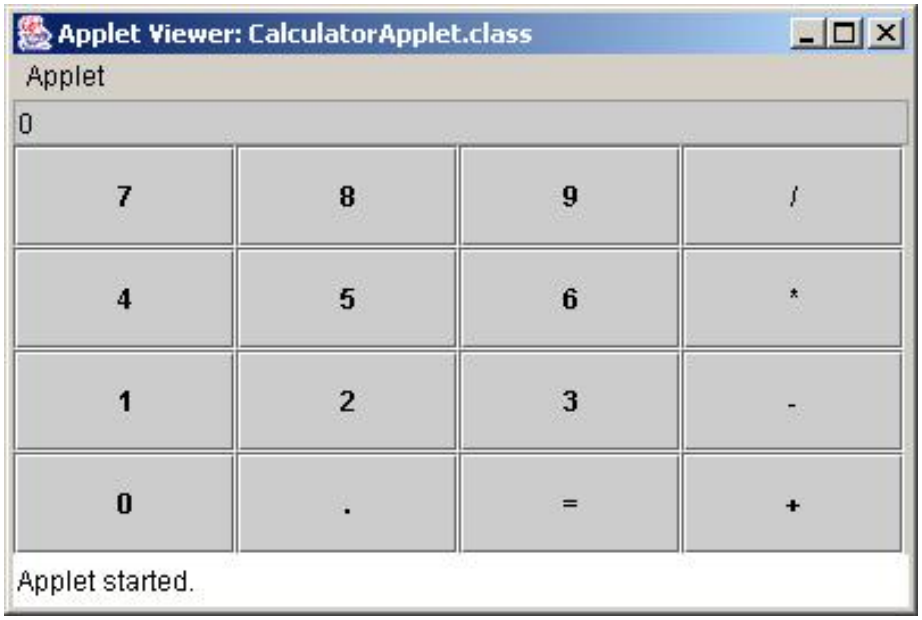


Figura 15.6 Saída com `Appletviewer`

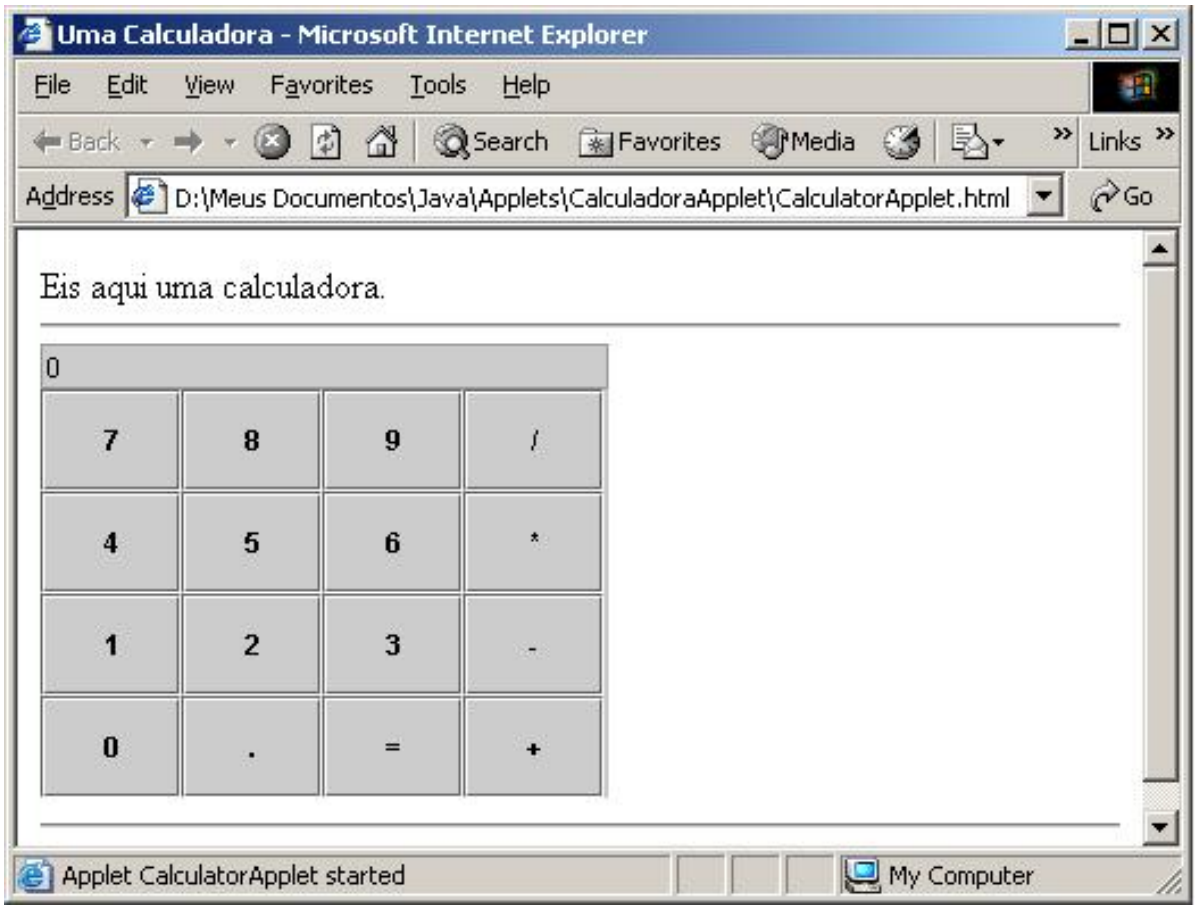


Figura 15.7 Saída com o Navegador

Capítulo 16

Objetos Remotos

Capítulo 17

Considerações Finais

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] CARDELLI, L. ; WEGNER, P. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17, (4):471-522, December 1985.

[4] CONELL, G ; HORSTMANN, C.S. & CAY S. *Core Java.* Makron Books, 1997.

[5] COAD, P. ; MAYFIELD, M. ; KERN, J. *Java Design: Building Better Apps and Applets*, Yourdon Press, 2nd Edition, 1999.

[6] COOPER, J. *Java Design Patterns: A Tutorial.* Addison-Wesley, 2000.

[7] DAHL, Ole-Johan ;NYGAARD, KRISTEN SIMULA: an ALGOL-based simulation language. Communications of the ACM, v.9 n.9, p.671-678, Sept. 1966

[8] DEITEL, H.M. ; DEITEL, P.J. *Java: Como Programar.* Sexta Edição. Pearson, Prentice-Hall, 2005.

[9] DESCARTES, R. *Discurso do Método: Regras para a Direção do Espírito*, Coleção A Obra-Prima de Cada Autor, Editora Martin Claret, 2000

[10] 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.

[11] DIJKSTRA, E. *A Discipline of Programming.* Prentice Hall, 1976.

[12] EGE, R. *Programming in a Object-Oriented Environment.* Academic Press, INC, San Diego, California, 1992.

[13] ELLIS, M. & STROUSTRUP, B. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[14] GAMMA, E.; HELM, R.;JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[15] GOLDBERG, A. & ROBSON, D. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.

[16] GOLDBERG, A. & ROBSON, D. *Smalltalk the language*. Addison-Wesley, 1989.

[17] HEHL, M.E. *Linguagem de Programação Estruturada FORTRAN 77*. McGraw-Hill, São Paulo, 1986.

[18] HOARE, C.A.R. *Editorial: The Quality of Software*. In *Software, Practice and Experience*, vol. 2, No 2, pages 103–105, 1972.

[19] HUGHES, J.K. *PL/I Structured Programming*. John Wiley & Sons, Inc, 1973.

[20] ICHBIAH, J.D. ; MORSER, S.P. *General Concepts of Simula 67 Programming Language*. 1967.

[21] JOHNSON, R.E. Frameworks=(Componentes + Patterns). *Communications of the ACM*, october 1997, vol. 40, No. 10.

[22] KERNIGHAN, B.; RITCHIE, D. *The C Programming Language (Ansi C)*. Prentice Hall Software Series, Second Edition, 1988.

[23] LADDAD, R. *AspectJ in Action: Practical Aspect Oriented Programming*. Manning Publications Co, 2003.

[24] LINTZ, B. P.; SWANSON, E.B. *Software Maintenance: A User/Management Tug of War* Data Management, pp. 26-30, april 1979.

[25] 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.

[26] LISKOV, B.; ZILLES, S. Programming with Abstract Data Type. *ACM Sigplan Notices*, march, 1974.

[27] LISKOV, B. et alii *CLU Reference Manual*. Springer-Verlag, Berlin, 1981.

[28] LISKOV, B. et alii *Abstraction and Specification in Program Development*. MIT Press and McGraw-Hill, New York, 1986.

[29] LISKOV, B. Data Abstraction and Hierarchy. *ACM Sigplan Notices*, 23,5 (May, 1988).

[30] MARTIN, R.C. *The Open-Closed Principle*. Disponível na página *www. objectmentor.com/resources/articles/ocp.pdf*. Acesso: 28/02/2005.

[31] MARTIN, R.C. *Design Principles and Design Patterns*. Disponível na página *www.objectmentor.com*. Acesso: 28/02/2005.

[32] MARTIN, R.C. *The Liskov Substitution Principles*. Disponível na página *www.objectmentor.com*. Acesso: 28/02/2005.

[33] MEYER, B. *Object-Oriented Software Construction*. Second Edition. Prentice Hall, 1997.

[34] MYERS, G. J. *Reliable Software Through Composite Design*. Petrocelli/Charter, 1975.

[35] MYERS, G. J. *Composite/Structured Design.* Van Nostrand Reinhold Company, 1978.

[36] 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.

[37] OSSHER, H.; TARR, P. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, 2001.

[38] PARNAS, D. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[39] RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language Reference Manual.* Addison-Wesley Longman, 1999.

[40] RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. *The Unified Modeling Language User Guide.* Addison-Wesley Longman, 1999.

[41] SCOTT, W. AMBLER. *Análise e Projeto Orientados a Objeto.* Volume 2, IBPI Press, Livraria e Editora Infobook S.A., 1998

[42] TIRELO, F. ; BIGONHA, R.S. ; VALENTE, M.T.O & BIGONHA, M.A.S. Programação Orientada por Aspectos, IN *JAI 2004*, SBC, Salvador.

[43] STRACHEY, CHRISTOPHER. *Fundamental concepts in programming languages.* Lecture notes for International Summer School in Computer Programming. Copenhagen, August 1967.

[44] STRACHEY, CHRISTOPHER. *Fundamental concepts in programming languages.* Higher-Order and Symbolic Computation. Copenhagen, April 2000.

[45] STROUSTRUP, BJARNE. *The C++ Programming Language.* Addison-Wesley, 1997.

[46] von STAA, A. *Programação Modular.* Editora Campus, 2000.

[47] WEXELBLAT, R.L. (ed.) *History of Programming Languages.* Academic Press, 1981, pp 25-74.

Índice Remissivo