

SANDRA PUGA • GERSON RISSETTI

Lógica de programação e estruturas de dados

com aplicações em Java



PEARSON
Prentice
Hall

www.prenhall.com/puga_br

Sandra Puga · Gerson Rissetti

Lógica de programação e estruturas de dados

com aplicações
em Java



São Paulo
Brasil Argentina Colômbia Costa Rica Chile
Espanha Guatemala México Peru Porto Rico Venezuela

© 2004 by Sandra Puga e Gerson Rissetti
Todos os direitos reservados. Nenhuma parte desta publicação
poderá ser reproduzida ou transmitida de qualquer modo
ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia,
gravação ou qualquer outro tipo de sistema de armazenamento e transmissão
de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

Diretor editorial: José Martins Braga
Editor: Roger Trimer
Editora de texto: Patrícia C. Rodrigues
Preparação: Maurício Kibe
Revisão: Thelma Guimarães e Juliana Takahashi
Capa: Marcelo Françozo
Projeto gráfico e diagramação: Figurativa Arte e Projeto Editorial

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

Puga, Sandra

Lógica de programação e estruturas de dados, com aplicações em Java /
Sandra Puga, Gerson Rissetti. -- São Paulo : Prentice Hall, 2003.

ISBN 85-87918-82-6

1. Dados - Estruturas (Ciência da computação)
2. Java (Linguagem de programação para computador).
3. Lógica I. Rissetti, Gerson. II. Título.

03-5239

CDD-005.1

Índices para catálogo sistemático:

1. Lógica estruturada : Computadores :
Processamento de dados 005.1

2004

Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil, uma empresa do grupo Pearson Education
Av. Ermano Marchetti, 1435, Lapa
CEP: 05038-001, São Paulo – SP, Brasil
Tel.: (11) 3613-1222. Fax: (11) 3611-0444
e-mail: vendas@pearsoned.com

A GRADECIMENTO ESPECIAL

Agradecemos aos professores e amigos Marcos Alberto Bussab e Robert Joseph Didio pela grande colaboração na tarefa de revisar e criticar o nosso livro.

SGP e GR

DEDICATÓRIA E A GRADECIMENTOS

Dedico este trabalho ao Pedro, pois este pequeno personagem foi quem me deu motivação para organizar e produzir este material.

Agradeço à Antonia por ter cuidado de mim e do Pedro durante as longas horas de trabalho.

SGP

SUMÁRIO

APRESENTAÇÃO	XI
INTRODUÇÃO	XIII
CAPÍTULO 1	
INTRODUÇÃO À LÓGICA.....	1
1.1 O USO DO RACIOCÍNIO LÓGICO NO DIA-A-DIA	2
1.2 O USO DA LÓGICA APLICADA À INFORMÁTICA	2
1.3 EXERCÍCIOS PARA FIXAÇÃO	4
1.4 EXERCÍCIOS COMPLEMENTARES	5
CAPÍTULO 2	
INTRODUÇÃO AOS ALGORITMOS	8
2.1 ALGORITMOS APLICADOS À SOLUÇÃO DE PROBLEMAS COMPUTACIONAIS	9
2.2 TIPOS DE ALGORITMOS	9
2.3 PSEUDOCÓDIGO	10
2.3.1 IDENTIFICAÇÃO DO ALGORITMO	10
2.3.2 DECLARAÇÃO DE VARIÁVEIS	11
2.3.3 CORPO DO ALGORITMO	12
2.4 FLUXOGRAMA	12
2.4.1 SIMBOLOGIA	12
2.5 EXERCÍCIOS PARA FIXAÇÃO	14
2.6 EXERCÍCIOS COMPLEMENTARES	15

CAPÍTULO 3

CONCEITOS BÁSICOS SOBRE ALGORITMOS	16
3.1 TIPOS DE DADOS	17
3.1.1 TIPOS CONSTRUÍDOS	19
3.2 VARIÁVEIS	19
3.2.1 IDENTIFICAÇÃO DAS VARIÁVEIS PARA OS ALGORITMOS	20
3.2.2 IDENTIFICADORES DE VARIÁVEIS PARA A LINGUAGEM JAVA	20
3.3 CONSTANTES	21
3.4 OPERADORES	21
3.4.1 OPERADORES DE ATRIBUIÇÃO	21
3.4.2 OPERADORES ARITMÉTICOS	22
3.4.3 OPERADORES RELACIONAIS	23
3.4.4 OPERADORES LÓGICOS	23
3.4.5 PRECEDÊNCIA DOS OPERADORES	23
3.5 TABELA-VERDADE	25
3.6 EXERCÍCIOS PARA FIXAÇÃO	26
3.7 EXERCÍCIOS COMPLEMENTARES	27

CAPÍTULO 4

CONCEITOS DE PROGRAMAÇÃO	29
4.1 INTRODUÇÃO À PROGRAMAÇÃO	30
4.2 TIPOS DE PROGRAMAÇÃO	32
4.2.1 CONCEITOS SOBRE A PROGRAMAÇÃO LINEAR	32
4.2.2 CONCEITOS SOBRE A PROGRAMAÇÃO ESTRUTURADA	32
4.3 CONCEITOS SOBRE A PROGRAMAÇÃO ORIENTADA A OBJETOS	34
4.3.1 O QUE É UM OBJETO	36
4.3.2 COMO VISUALIZAR UM OBJETO?	37
4.3.3 CONCEITO DE CLASSES	37
4.3.4 INSTÂNCIAS DE OBJETOS	38
4.3.5 HERANÇA	39
4.3.6 POLIMORFISMO	39
4.3.7 ENCAPSULAMENTO	39
4.3.8 APLICAÇÃO	41
4.4 EXERCÍCIOS PARA FIXAÇÃO	45
4.5 EXERCÍCIOS COMPLEMENTARES	45

CAPÍTULO 5

CONSTRUÇÃO DE ALGORITMOS: ESTRUTURAS DE CONTROLE	46
5.1 ENTRADA	47

5.2	SAÍDA	48
5.3	ESTRUTURAS DE SELEÇÃO OU DECISÃO	53
5.4	ESTRUTURAS DE SELEÇÃO SIMPLES	54
5.5	ESTRUTURAS DE SELEÇÃO COMPOSTAS	56
5.6	ESTRUTURAS DE SELEÇÃO ENCADEADAS	58
5.7	ESTRUTURAS DE SELEÇÃO DE MÚLTIPLA ESCOLHA	62
5.8	ESTRUTURAS DE REPETIÇÃO	66
5.8.1	ESTRUTURA DE REPETIÇÃO COM TESTE NO INÍCIO – ESTRUTURA ENQUANTO	66
5.8.2	ESTRUTURA DE REPETIÇÃO COM TESTE NO FIM – ESTRUTURA REPITA	69
5.8.3	ESTRUTURA DE REPETIÇÃO COM VARIÁVEL DE CONTROLE – ESTRUTURA PARA	72
5.9	EXERCÍCIOS PARA FIXAÇÃO	74
5.10	EXERCÍCIOS SUGERIDOS	75
5.11	EXERCÍCIOS COMPLEMENTARES	76

CAPÍTULO 6

	ESTRUTURA DE DADOS: VETORES	78
6.1	ESTRUTURAS INDEXADAS – VETOR (ARRAY)	79
6.1.1	DECLARAÇÃO DE VETOR	79
6.1.2	ACESSO E ATRIBUIÇÃO DE VALOR EM VETORES	81
6.1.3	OPERAÇÕES	86
6.2	CONCEITO DE MATRIZES	95
6.2.1	DECLARAÇÃO	96
6.2.2	OPERAÇÕES	96
6.3	EXERCÍCIOS PARA FIXAÇÃO	102
6.4	EXERCÍCIOS COMPLEMENTARES	104

CAPÍTULO 7

	PROCEDIMENTOS E FUNÇÕES	105
7.1	PROCEDIMENTOS	106
7.1.1	CHAMADA DE PROCEDIMENTOS	108
7.2	FUNÇÕES	113
7.3	ESCOPO DE VARIÁVEIS	115
7.4	PARÂMETROS	116
7.4.1	PARÂMETROS FORMAIS	116
7.4.2	PARÂMETROS REAIS	117
7.4.3	PASSAGEM DE PARÂMETROS	119
7.4.4	PASSAGEM DE PARÂMETROS POR VALOR	119
7.4.5	PASSAGEM DE PARÂMETROS POR REFERÊNCIA	120
7.5	EXERCÍCIOS PARA FIXAÇÃO	120
7.6	EXERCÍCIOS COMPLEMENTARES	120

CAPÍTULO 8

BUSCA E ORDENAÇÃO	122
8.1 ORDENAÇÃO POR TROCAS — MÉTODO DA BOLHA	123
8.2 BUSCA	131
8.2.1 BUSCA LINEAR (OU SEQÜENCIAL)	131
8.2.2 BUSCA BINÁRIA (OU BUSCA LOGARÍTMICA)	137
8.3 EXERCÍCIOS PARA FIXAÇÃO	141
8.4 EXERCÍCIOS COMPLEMENTARES	142

CAPÍTULO 9

ACESSO A ARQUIVOS	143
9.1 O QUE É UM ARQUIVO?	144
9.2 ARQUIVO-TEXTO	144
9.3 TIPOS DE ARQUIVO QUANTO ÀS FORMAS DE ACESSO	145
9.3.1 ARQUIVOS SEQÜENCIAIS	145
9.3.2 ARQUIVOS DE ACESSO ALEATÓRIO	146
9.4 OPERAÇÕES DE MANIPULAÇÃO DE ARQUIVOS	146
9.4.1 REPRESENTAÇÃO DA MANIPULAÇÃO DE ARQUIVOS SEQÜENCIAIS	146
9.4.2 REPRESENTAÇÃO DA MANIPULAÇÃO DE ARQUIVOS DE ACESSO ALEATÓRIO	146
9.5 EXERCÍCIOS PARA FIXAÇÃO	180
9.6 EXERCÍCIOS COMPLEMENTARES	181

CAPÍTULO 10

ESTRUTURAS DE DADOS DINÂMICAS	182
10.1 LISTAS	183
10.1.1 LISTAS ENCADEADAS	184
10.1.2 TIPOS DE LISTAS ENCADEADAS	186
10.2 LISTAS DE ENCADEAMENTO SIMPLES	187
10.3 LISTAS DUPLAMENTE ENCADEADAS	201
10.4 FILAS	205
10.5 PILHAS	213
10.6 ÁRVORES	220
10.6.1 ÁRVORES BINÁRIAS	221
10.7 EXERCÍCIOS PARA FIXAÇÃO	233

ANEXO

CONCEITOS SOBRE A LINGUAGEM JAVA	236
---	------------

APRESENTAÇÃO

Um livro que aliasse a lógica de programação, que é efetivamente praticada nos meios acadêmicos, com sua implementação na linguagem Java era um anseio de professores e alunos, e é a esse anseio que esta publicação vem atender.

O livro aborda desde os princípios básicos de lógica até assuntos mais complexos, mas igualmente necessários, numa linguagem que, sem perder o rigor científico necessário às obras desse porte, é comprehensível pelos estudantes de diferentes graus de conhecimento e dificuldade. Os tópicos vão desfilando de maneira didática e tecnicamente adequada, tornando o aprendizado uma tarefa menos árdua. O texto é rico em exemplos e há uma série de exercícios práticos com diferentes graus de dificuldade.

Esta obra é, acima de tudo, um reflexo da experiência profissional e acadêmica de seus autores e uma contribuição de suma importância para a bibliografia nacional na área de computação e informática.

Prof. Marcos Alberto Bussab
Coordenador da área de ciências da computação da Uninove

INTRODUÇÃO

Lógica de programação e estruturas de dados com aplicações em Java é um livro destinado a todas as pessoas interessadas em programação de computadores, mas especialmente aos estudantes da área de computação, pois foram eles nossa grande fonte de inspiração — muitos dos exemplos e exercícios são frutos colhidos nas salas de aula.

Procuramos sintetizar num único livro os assuntos que julgamos essenciais para contribuir para a formação de um bom programador: lógica de programação, estrutura de dados e aplicações em Java. Escolhemos a linguagem de programação Java para implementação dos algoritmos por dois motivos: a necessidade de profissionais especializados em Java e a necessidade de uma bibliografia básica que mostrasse passo a passo o uso de estruturas de seleção, repetição, ordenação, filas e pilhas, dentre outros, implementadas em Java.

A linguagem de programação Java é fundamentalmente orientada a objetos, mas nosso objetivo não é estudar esse tipo de programação. Faremos uso de recursos bastante simples, sem nos preocuparmos com questões mais sofisticadas da orientação a objetos, que serão levemente abordadas no Capítulo 4.

Os conceitos aqui abordados poderão ser adaptados às questões encontradas no dia-a-dia, inclusive com o uso de outras linguagens, uma vez que todos os exemplos são descritos em pseudocódigo e depois transcritos para Java.

No Capítulo 1 são abordados alguns conceitos da lógica como ciência pura, ainda sem a sua aplicação na computação. Sem grandes pretensões, queremos apenas mostrar que o uso da lógica faz parte de nosso cotidiano.

No Capítulo 2 mostramos algumas das aplicações dos algoritmos na resolução de diferentes tipos de problemas. É apresentada uma breve introdução ao conceito de entrada, processamento e saída e também são apresentados os tipos de algoritmos pseudocódigo e fluxograma.

No Capítulo 3 são apresentados os tipos de dados básicos e seus desdobramentos na linguagem de programação Java. Além disso, são definidos o conceito, a aplicação e a identificação de variáveis e constantes e é demonstrado o uso dos operadores de atribuição, aritméticos, relacionais e lógicos tanto na notação algorítmica como na linguagem de programação Java. Nesse capítulo também são exemplificados a construção de expressões de atribuição, aritméticas e lógicas, a ordem de precedência matemática utilizada na resolução de problemas e o uso da tabela-verdade como recurso facilitador do entendimento do uso dos operadores lógicos.

O Capítulo 4 aborda de maneira bastante simplificada alguns dos paradigmas da programação: linear, estruturada e orientada a objetos.

No Capítulo 5 — que é aquele que trata dos conceitos relacionados à lógica de programação — são estudados os recursos para entrada e saída de dados e o uso de estruturas de repetição e seleção. É por meio do uso dessas estruturas que o programador cria rotinas para controle do fluxo dos dados em seus programas — rotinas que possibilitem o acesso ou não a determinadas informações e que implementem estruturas de dados mais sofisticadas.

O Capítulo 6 trata das estruturas de dados estáticas e homogêneas, isto é, dos vetores e matrizes, e das operações que estas suportam. Também são abordadas algumas aplicações práticas.

No Capítulo 7 são abordados alguns recursos que melhoram a legibilidade do código de programação ou do algoritmo e que podem possibilitar a reutilização do código por outros programas; estes recursos são os procedimentos, as funções e os parâmetros que são passados para eles.

No Capítulo 8 são apresentados recursos para facilitar a ordenação das informações que serão armazenadas ou manipuladas e para possibilitar a busca de informações específicas.

O Capítulo 9 trabalha as técnicas de criação e manipulação de arquivos-texto seqüenciais e randômicos.

O Capítulo 10 aborda as estruturas de dados, pilhas, filas e árvores de maneira simples e com exemplos que ilustram, passo a passo, como criar e utilizar essas estruturas.

Por último, no anexo, são apresentados alguns recursos do Java.

Assim, este livro procura de maneira bastante simples abordar algumas questões que, por vezes, parecem muito complexas ao programador iniciante; todos os assuntos são explicados e exemplificados por meio de soluções comentadas. Ao final de cada

capítulo existem exercícios propostos, os quais tem como objetivo fixar o conteúdo estudado ou então, no caso de exercícios mais sofisticados, complementar o aprendizado. No site da editora estão disponíveis outros exercícios complementares e desafios.

Esperamos, com este livro, poder contribuir para o aprendizado dos nossos leitores.

Sandra Gavioli Puga e Gerson Rissetti

INTRODUÇÃO À LÓGICA

- ▶ *Introdução à lógica*
- ▶ *Aplicações da lógica*
- ▶ *Exercícios para fixação*
- ▶ *Exercícios complementares*



OBJETIVOS:

Abordar o conceito de lógica como ciência; destacar o uso da lógica de maneira muitas vezes incondicional, nas tarefas do dia-a-dia; usar o raciocínio lógico para a tomada de decisões e para a resolução de problemas.

O filósofo grego Aristóteles é considerado o criador da lógica. No entanto, ele não a chamava assim, denominava-a ‘razão’. O termo ‘lógica’ só passou a ser utilizado mais tarde.

A palavra ‘lógica’ é originária do grego *logos*, que significa linguagem racional. De acordo com o dicionário *Michaelis*, lógica é a análise das formas e leis do pensamento, mas não se preocupa com a produção do pensamento, quer dizer, não se preocupa com o conteúdo do pensamento, mas sim com a forma deste, isto é, com a maneira pela qual um pensamento ou uma idéia são organizados e apresentados, possibilitando que cheguemos a uma conclusão por meio do encadeamento dos argumentos.

Os argumentos podem ser dedutivos ou indutivos. Os argumentos indutivos são aqueles com que, a partir dos dados, se chega a uma resposta por meio da analogia, ou

seja, pela comparação com algo conhecido, porém esse tipo de raciocínio não oferece certeza de que a resposta será de fato verdadeira. É necessário conhecer os fatos ou as situações para que se possa fazer a comparação. Por exemplo: ontem não havia nuvens no céu e não choveu. Hoje não há nuvens no céu, portanto não vai chover.

Já os argumentos dedutivos são aqueles cuja conclusão é obtida como consequência das premissas, isto é, por meio da análise das situações ou fatos, pode-se obter a resposta. Trabalha-se com a forma das sentenças, sem que haja necessidade do conhecimento prévio das situações ou fatos. Por exemplo: Joana é uma mulher. As mulheres são seres humanos. Logo, Joana é um ser humano.

A lógica nos permite caminhar pelos limites das diversas ciências!

1.1 O USO DO RACIOCÍNIO LÓGICO NO DIA-A-DIA

Desde os tempos primitivos o homem utiliza-se do raciocínio lógico para a realização das suas atividades. Isso é comprovado pelo fato de ele ter estabelecido seqüências adequadas para a realização das suas tarefas com sucesso. Podemos citar alguns exemplos relacionados às suas atividades do dia-a-dia:

- Uma pessoa adulta, para tomar banho, primeiro tira a roupa para não molhá-la e também para estabelecer contato direto entre sua pele e a água.
- Uma criança, desde pequenina, aprende que, para chupar uma bala, é preciso tirá-la da embalagem.
- Foi utilizando-se do raciocínio lógico que o homem conseguiu criar a roda!

1.2 O USO DA LÓGICA APLICADA À INFORMÁTICA

A lógica é aplicada a diversas ciências, tais como a informática, a psicologia e a física, entre outras. Na informática e na computação, aplica-se a todas as suas áreas, para a construção e funcionamento do hardware e do software. Por exemplo, na construção de um circuito integrado para o teclado, trabalha-se com o conceito de portas lógicas para a verificação da passagem ou não de pulsos elétricos, a fim de que seja estabelecida uma comunicação entre os componentes. Já na construção de software, é por meio do raciocínio lógico que o homem constrói algoritmos que podem ser transformados em programas de computador capazes de solucionar problemas cada vez mais complexos. É justamente esse assunto que estudaremos neste livro.

NOTA:

Hardware – Parte física do computador – peças. Exemplo: teclado.

Software – Parte lógica do computador – programas. Exemplo: Windows.

Algoritmo – Seqüência de passos ordenados para a realização de uma tarefa.

Programa – Conjunto de instruções legíveis para o computador e capazes de realizar tarefas.

Para nos auxiliar na resolução dos problemas de construção de algoritmos aplicados à informática, faremos uso da lógica formal dedutiva. No entanto, para que sejam reunidos dados para a solução dos problemas, muitas vezes utilizaremos o raciocínio lógico indutivo.

Como foi visto anteriormente, a lógica preocupa-se com a forma da construção do pensamento. Isso permite que se trabalhe com variáveis para que se possa aplicar o mesmo raciocínio a diferentes problemas. Por exemplo:

Gerson é cientista.

Todo cientista é estudioso.

Logo, Gerson é estudioso.

Substituindo as palavras ‘Gerson’ e ‘estudioso’ por A e B:

A é cientista.

Todo cientista é B.

Logo, A é B.

NOTA:

Variáveis – Vide o Capítulo 3.

O raciocínio lógico nos conduz a uma resposta que pode ser ‘verdadeiro’ ou ‘falso’. Na construção de algoritmos para a solução de problemas computacionais, trabalha-se com esse tipo de raciocínio. As informações a ser analisadas são representadas por variáveis que posteriormente receberão valores. As variáveis, por sua vez, representarão as premissas. Por exemplo:

Dados dois valores quaisquer, deseja-se saber qual é o maior.

Os dois valores são representados pelas variáveis A e B. Analisa-se o problema a fim de averiguar qual é a melhor maneira de descobrir a solução, então se monta a seqüência para que seja verificada a questão. Para descobrir a solução, pode-se partir de problemas similares já resolvidos e, por analogia, aplicar o mesmo método ao problema atual, ou podem-se estudar formas de resolvê-lo buscando dados com especialistas no assunto em questão.

Nesse caso, vamos substituir as variáveis por valores conhecidos, apenas como modelo para facilitar o entendimento do raciocínio aplicado:

A será substituída por 7 e B, por 19.

Para que seja verificado o maior valor, deve-se fazer uma comparação, por exemplo: 7 é maior do que 19?

Logo tem-se a resposta: falso.

Então, pode-se concluir que 19 é o maior número entre os dois.

LEMBRE-SE:

A resposta a uma questão deve ser ‘verdadeiro’ ou ‘falso’; nunca pode ser as duas opções ao mesmo tempo.

Quando os valores são desconhecidos, na representação para a solução do problema, trabalha-se apenas com as variáveis:

A é maior do que B?

Se a resposta é ‘verdadeiro’, A é o maior valor.

Se a resposta é ‘falso’, B é o maior valor.

NOTA:

Não está sendo considerada a possibilidade de os valores de A e B serem iguais, por se tratar apenas de um exemplo para a construção do raciocínio, não sendo levada em conta a complexidade do problema em questão para o caso de uma implementação.

1.3 EXERCÍCIOS PARA FIXAÇÃO

1. Dadas as premissas a seguir, verifique quais são as sentenças que representam a conclusão correta:

I – Cavalos são animais. Animais possuem patas. Logo:

- a) Cavalos possuem patas.
- b) Todos os animais são cavalos.
- c) Os cavalos possuem quatro patas.

II – Retângulos são figuras que têm ângulos. Temos uma figura sem nenhum ângulo. Logo:

- a) Essa figura pode ser um círculo.
- b) Não é possível tirar conclusões.
- c) Essa figura não é um retângulo.

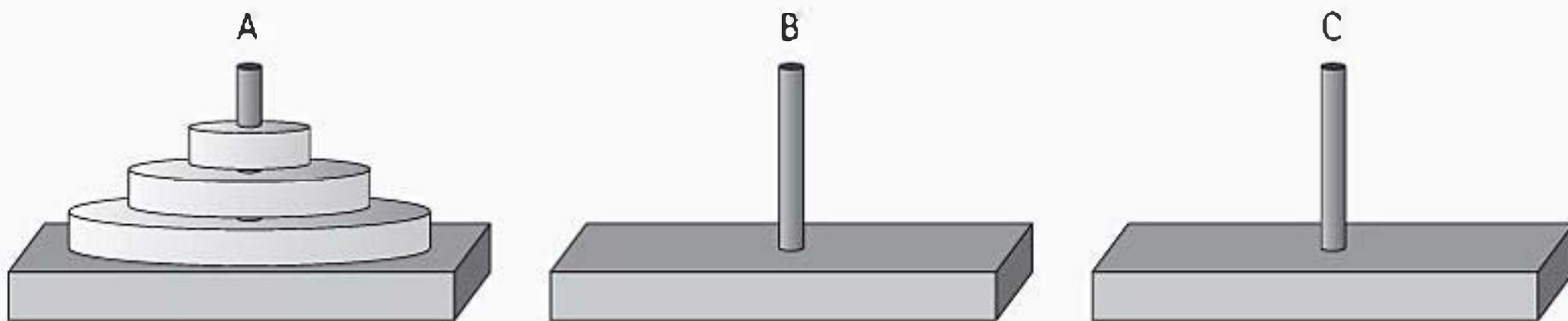
III – Se o verde é forte, o vermelho é suave. Se o amarelo é suave, o azul é médio. Mas ou o verde é forte ou o amarelo é suave. Forte, suave e médio são as únicas tonalidades possíveis. Logo:

- a) O azul é médio.
- b) Ou o vermelho é suave ou o azul é médio.
- c) O amarelo e o vermelho são suaves.

2. Responda:

- a) Qual é a importância da lógica para a informática?
- b) Descreva algumas atividades relacionadas ao seu dia-a-dia nas quais o uso da lógica se faz presente e perceptível.
- c) O que são argumentos?
- d) Qual é a diferença entre argumentos dedutivos e argumentos indutivos? Exemplifique.

3. Analise e descreva uma maneira de mover os discos do pino A para o pino C, mantendo a mesma ordem. Em hipótese nenhuma um disco maior poderá ficar sobre um menor. Para que um disco seja movido de A para C, deve-se passar pelo pino B e vice-versa.



1.4 EXERCÍCIOS COMPLEMENTARES

1. Dadas as premissas a seguir, verifique quais são as sentenças que representam a conclusão correta:

I – Você está dirigindo seu carro. Se brecar repentinamente, um caminhão baterá na traseira dele. Se não brecar imediatamente, você atropelará uma criança que está atravessando a estrada. Logo:

- a) As crianças devem afastar-se das estradas.
- b) O caminhão baterá na traseira de seu carro ou você atropelará a criança.
- c) O caminhão vai muito depressa.

II – Somente quando B é X, K é Z. E é X ou Z somente quando K não é Z. Duas letras não podem ser uma só. Logo:

- a) Quando B é X, E não é X nem Z.
- b) Quando K é Z, X ou Z é E.
- c) Quando B não é X, E não é X nem Z.

III – Quando B é maior que A, J é menor que A. Porém, A nunca é maior que B e jamais é igual a B. Logo:

- a) J nunca é menor que B.
- b) J nunca é menor que A.
- c) J nunca é maior que B.

IV – Todas as plantas verdes têm clorofila. Algumas coisas que têm clorofila são comestíveis. Logo:

- a) Alface é comestível.
- b) Algumas plantas verdes são comestíveis.
- c) Alface tem clorofila.

2. De acordo com as pistas de cada uma das histórias, descubra o que se pede:
- a) As amigas de Maria organizaram um chá-de-panela para comemorar seu casamento, que estava próximo. Como é de costume, cada amiga compareceu à reunião com um presente devidamente embrulhado. O chá-de-panela consiste em uma brincadeira que é feita com a noiva, na qual ela deve adivinhar o presente contido em cada embrulho que recebe. Se errar, recebe castigos. Sua tarefa é descobrir o presente que cada amiga levou. De acordo com as dicas abaixo, preencha a tabela.
- Maria adivinhou os presentes de Janete e Sandra.
 - Maria não adivinhou o conteúdo do embrulho que continha uma garrafa térmica, por isso teve de vestir uma fantasia de coelhinha.
 - Márcia pediu que Maria dançasse a dança da garrafa.
 - Renata a castigou com uma maquiagem de palhacinho.
 - Maria aceitou os embrulhos da frigideira e da jarra para suco.
 - O faqueiro não foi presente de Izabel.
 - Por ter errado o caldeirão, Maria acabou ficando embriagada.
 - No embrulho de Sandra estava escrito ‘frágil’ e isso facilitou muito a descoberta.
- | Colega | Presente |
|--------|----------|
| | |
| | |
| | |
| | |
| | |
- b) Oito carros de equipes diferentes estão alinhados lado a lado para uma corrida. De acordo com as pistas abaixo, descubra a ordem dos carros para a largada e a cor de cada carro. (Obs.: a cor utilizada não é a cor original das equipes.)
- O carro branco está à esquerda do Lotus.
 - O carro da equipe Ferrari está entre os carros vermelho e branco.
 - O McLaren é o segundo carro à esquerda do Ferrari e o primeiro à direita do carro azul.
 - O Sauber não tem carro à sua direita e está logo depois do carro preto.
 - O carro preto está entre o Sauber e o carro amarelo.

- O Jaguar não tem carro algum à sua esquerda e está à esquerda do carro verde.
 - À direita do carro verde está o Renault.
 - O Jordan é o segundo carro à direita do carro prata e o segundo à esquerda do carro laranja.
 - O Toyota é o segundo carro à esquerda do Minardi.
3. Um pastor deve levar suas três ovelhas e seus dois lobos para o pasto que fica ao sul da região. Ele deve levar também a provisão de alimentos para as ovelhas, que consiste em dois maços de feno. No entanto, no meio do caminho existe um grande rio cheio de piranhas e o pastor tem apenas um pequeno barco à sua disposição, que lhe permite levar dois ‘passageiros’ de cada vez. Considere como passageiros as ovelhas, os maços de feno e os lobos e considere que, se as ovelhas ficarem em menor número do que os lobos, serão comidas e que, se o feno ficar com as ovelhas sem um lobo por perto, as ovelhas comerão o feno. Ajude o pastor a atravessar o rio e preservar suas posses.

2

INTRODUÇÃO AOS ALGORITMOS

- ▶ *Introdução aos algoritmos*
- ▶ *Tipos de algoritmos*
- ▶ *Pseudocódigo*
- ▶ *Fluxograma*
- ▶ *Exercícios para fixação*
- ▶ *Exercícios complementares*

OBJETIVOS:



Mostrar as aplicações dos algoritmos para resolução de diferentes problemas; especificar a importância dos algoritmos para a resolução de problemas computacionais; abordar os conceitos de entrada, processamento e saída do ponto de vista computacional; definir os tipos de algoritmos a serem utilizados neste livro (pseudocódigo e fluxograma).

Um algoritmo é uma seqüência lógica de instruções que devem ser seguidas para a resolução de um problema ou para a execução de uma tarefa. Os algoritmos são amplamente utilizados nas disciplinas ligadas à área de ciências exatas, tais como matemática, física, química e informática, entre outras, e também são utilizados com muito sucesso em outras áreas.

No dia-a-dia, as pessoas utilizam-se de algoritmos de maneira intuitiva, sem que haja necessidade de planejar previamente a seqüência de passos para a resolução das tarefas diárias. Dentre os inúmeros exemplos existentes, podemos citar:

1. Quando uma dona de casa prepara um bolo, segue uma receita, que nada mais é do que um algoritmo em que cada instrução é um passo a ser seguido para que o prato fique pronto com sucesso:

Bata quatro claras em neve.

Adicione duas xícaras de açúcar.

Adicione duas xícaras de farinha de trigo, quatro gemas, uma colher de fermento e duas colheres de chocolate.

Bata por três minutos.

Unte uma assadeira com margarina e farinha de trigo.

Coloque o bolo para assar durante vinte minutos em temperatura média.

2. Um motorista que necessita efetuar a troca de um pneu furado segue uma rotina para realizar essa tarefa:

Verifica qual pneu está furado.

Posiciona o macaco para levantar o carro.

Pega o estepe.

Solta os parafusos.

Substitui o pneu furado.

Recoloca os parafusos.

Desce o carro.

Guarda o macaco e o pneu furado.

3. Um matemático, para resolver uma equação qualquer, utiliza passos pré-determinados que conduzem à obtenção do resultado.

2.1 ALGORITMOS APLICADOS À SOLUÇÃO DE PROBLEMAS COMPUTACIONAIS

Os algoritmos são amplamente utilizados na área da computação, seja na elaboração de soluções voltadas à construção de interfaces, *software* e *hardware*, seja no planejamento de redes. Os algoritmos também constituem uma parte importante da documentação de sistemas, pois descrevem as tarefas a serem realizadas pelos programas.

2.2 TIPOS DE ALGORITMOS

Existem diversos tipos de algoritmos. Dentre eles, podemos citar: pseudocódigo, descrição narrativa, fluxograma e diagrama de Chapin.

- O **pseudocódigo** utiliza linguagem estruturada e se assemelha, na forma, a um programa escrito na linguagem de programação Pascal. O pseudocódigo é também denominado por alguns autores como **português estruturado**, embora existam pequenas diferenças de metodologia entre ambos. É bastante utilizado para representação da resolução de problemas computacionais.
- A **descrição narrativa** utiliza linguagem natural para especificar os passos para a realização das tarefas. Isso dá margem a más interpretações e ambigüidades. Não é muito utilizada.
- O **fluxograma** é uma forma universal de representação, pois se utiliza de figuras geométricas para ilustrar os passos a serem seguidos para a resolução dos problemas. Bastante utilizado, é também chamado por alguns autores de **diagrama de blocos**.
- O **diagrama de Chapin**, também conhecido como **diagrama Nassi-Shneiderman** ou **diagrama N-S**, apresenta a solução do problema por meio de um diagrama de quadros com uma visão hierárquica e estruturada. Esse tipo de diagrama não é muito utilizado, pois é muito difícil representar recursividade, entre outros procedimentos.

Neste livro, serão abordadas as duas formas mais comuns de representação de soluções para problemas computacionais: pseudocódigo e fluxograma. A seguir, serão descritas as características de cada um.

2.3 PSEUDOCÓDIGO

O **pseudocódigo** é um tipo de algoritmo que utiliza uma linguagem flexível, intermediária entre a linguagem natural e a linguagem de programação. É utilizado para organizar o raciocínio lógico a ser seguido para a resolução de um problema ou para definir os passos para a execução de uma tarefa. É também utilizado para documentar rotinas de um sistema.

A palavra ‘pseudocódigo’ significa ‘falso código’. Esse nome se deve à proximidade que existe entre um algoritmo escrito em pseudocódigo e a maneira pela qual um programa é representado em uma linguagem de programação.

O Exemplo 2.1 propõe um problema simples para o qual será desenvolvido o algoritmo em pseudocódigo.

2.3.1 IDENTIFICAÇÃO DO ALGORITMO

Todo algoritmo representado por um pseudocódigo deverá ser, primeiramente, identificado. Para se identificar ou nomear o algoritmo, recomenda-se:

- Não utilizar espaços entre as letras. Por exemplo: para um cadastro de clientes, o correto seria `cad_cli` ou `cadcliente`. O caractere ‘sublinha’ ou ‘underline’ (_) pode ser utilizado para representar o espaço entre as letras.

EXEMPLO 2.1: Desenvolver um pseudocódigo para ler o nome, a idade, o cargo e o salário de 50 pessoas e verificar quantas possuem idade inferior a 30 anos e um salário superior a R\$ 3.000,00.

Algoritmo Exemplo_2.1

Identificação do algoritmo

Var nome, cargo: literal
idade, n_pessoas, tot_pessoas: inteiro
salario: real

Declaração das variáveis

Início

Corpo do algoritmo

n_pessoas ← 1
tot_pessoas ← 0

Enquanto (n_pessoas <= 50) Faça
 Ler (nome, idade, cargo, salario)
 Se (idade <= 30) e (salario >= 3000,00) Então
 tot_pessoas ← tot_pessoas + 1
 Fim-Se
 n_pessoas ← n_pessoas + 1
Fim-Enquanto
Mostrar ("• total de pessoas que atendem a condição é",
tot_pessoas)
Fim.

- Não iniciar o nome com algarismos (números). Por exemplo: não usar 1algoritmo. O correto seria algoritmo1.
- Não utilizar palavras reservadas, isto é, palavras que são utilizadas nos algoritmos para representar ações específicas. Por exemplo: se (palavra que representa uma condição ou teste lógico); var (palavra que representa a área de declaração de variáveis).
- Não utilizar caracteres especiais, como acentos, símbolos (? / : @# etc.), ç, entre outros.
- Não utilizar nomes iguais para representar variáveis diferentes.
- Ser sucinto e utilizar nomes coerentes.

2.3.2 DECLARAÇÃO DE VARIÁVEIS

Todas as variáveis que serão utilizadas na resolução do problema devem ser previamente declaradas, isto é, todas as informações necessárias à resolução do problema devem ser representadas. Este assunto será abordado com mais detalhes no Capítulo 3.

2.3.3 CORPO DO ALGORITMO

É a área do algoritmo reservada para a resolução do problema. Nessa parte, devem-se escrever todos os passos lógicos necessários para solucionar o problema, tais como:

- representar a entrada de valores para as variáveis;
- representar as operações de atribuição, lógicas e aritméticas;
- representar a abertura e fechamento de arquivos;
- representar os laços de repetição;
- representar a exibição dos resultados; entre outros.

2.4 FLUXOGRAMA

O **fluxograma** é um tipo de algoritmo que utiliza símbolos gráficos para representar as ações ou instruções a serem seguidas. Assim como o pseudocódigo, o fluxograma é utilizado para organizar o raciocínio lógico a ser seguido para a resolução de um problema ou para definir os passos para a execução de uma tarefa. Também é utilizado para documentar rotinas de um sistema, mas só é recomendado para casos pouco extensos.

NOTA:

● *fluxograma, por utilizar figuras para representação das ações, é considerado um algoritmo universal.*

2.4.1 SIMBOLOGIA

Cada instrução ou ação a ser executada deve ser representada por meio de um símbolo gráfico. Os símbolos utilizados neste livro são apresentados a seguir:



Terminal

Representa o início e o final do fluxograma.



Processamento

Representa a execução de operações ou ações como cálculos aritméticos, atribuição de valores a variáveis, abertura e fechamento de arquivo, entre outras.



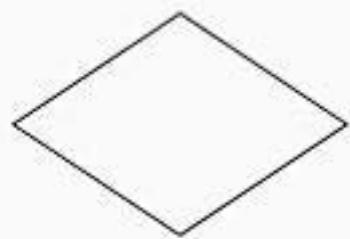
Teclado

Representa a entrada de dados para as variáveis por meio do teclado.



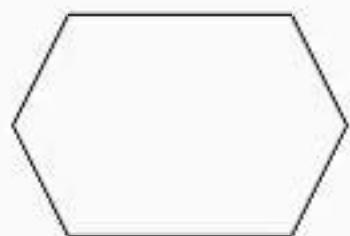
Vídeo

Representa a saída de informações (dados ou mensagens) por meio do monitor de vídeo ou outro dispositivo visual de saída de dados.



Decisão

Representa uma ação lógica que resultará na escolha de uma das seqüências de instruções, ou seja, se o teste lógico apresentar o resultado ‘verdadeiro’, realizará uma seqüência e, se o teste lógico apresentar o resultado ‘falso’, realizará outra seqüência.



Preparação

Representa uma ação de preparação para o processamento, ou seja, um processamento pré-definido.



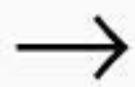
Conector

Utilizado para interligar partes do fluxograma ou para desviar o fluxo corrente para um determinado trecho do fluxograma.



Conector de páginas

Utilizado para interligar partes do fluxograma em páginas distintas.

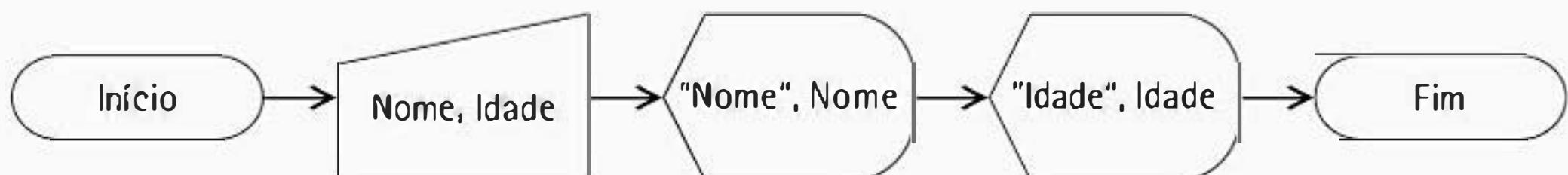


Seta de orientação do fluxo

A seqüência do fluxograma pode ser desenvolvida horizontalmente ou verticalmente.

Para o Exemplo 2.2, apresentado a seguir, o fluxograma foi feito horizontalmente e representa a entrada de dados por meio do teclado e a saída pelo vídeo dos dados inseridos.

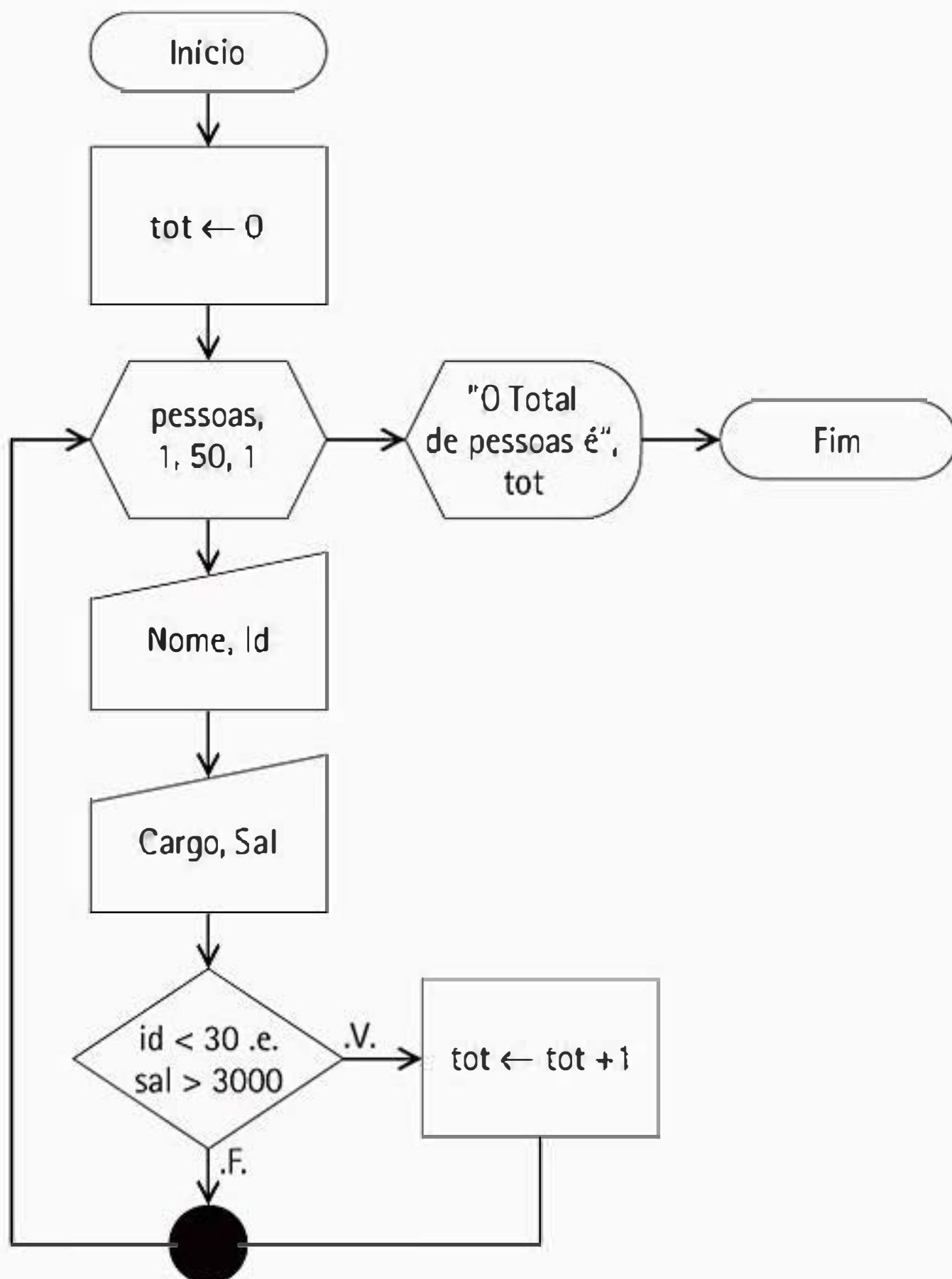
EXEMPLO 2.2: Ler o nome e a idade de uma pessoa e mostrar na tela.



|FIGURA 1| Fluxograma — sentido horizontal

Já para o Exemplo 2.3, o fluxograma foi desenvolvido seguindo basicamente a orientação vertical. Esse exemplo, mais complexo do que o anterior, utiliza-se de laços de repetição e seleção (que serão tratados no Capítulo 5) e processamento, além da entrada de dados pelo teclado e saída de dados pelo vídeo. Pode-se observar, também, a presença do conector.

EXEMPLO 2.3: Desenvolver um fluxograma para ler o nome, a idade, o cargo e o salário de 50 pessoas e verificar quantas têm idade inferior a 30 anos e um salário superior a R\$ 3.000,00.



|FIGURA 2| Fluxograma — sentido vertical

2.5 EXERCÍCIOS PARA FIXAÇÃO

1. Qual é a principal função dos algoritmos?
2. Descreva três tipos de algoritmos.
3. Comparando-se o fluxograma ao pseudocódigo, pode-se perceber que, no fluxograma, as variáveis não precisam ser declaradas. No entanto, existe uma similaridade na seqüência de resolução das tarefas em ambos. Observe qual é a similaridade e comente-a.
4. Escreva a seqüência de passos para que um robô seja capaz de trocar uma lâmpada queimada que está localizada no centro de uma sala. Há uma escada posicionada logo abaixo da lâmpada queimada e o robô está em frente à escada.

2.6 EXERCÍCIOS COMPLEMENTARES

1. Escreva a seqüência de passos para que uma pessoa abra um arquivo armazenado em um disquete utilizando o Word for Windows.
2. Escreva os passos necessários para uma pessoa efetuar um saque em um caixa eletrônico.
3. Escreva os passos necessários para uma pessoa efetuar uma compra por meio da Internet.

CONCEITOS BÁSICOS SOBRE ALGORITMOS

- ▶ *Tipos de dados*
- ▶ *Variáveis*
- ▶ *Constantes*
- ▶ *Operadores*
- ▶ *Tabela-verdade*
- ▶ *Exercícios para fixação*
- ▶ *Exercícios complementares*

OBJETIVOS:

Mostrar os tipos de dados básicos e seus desdobramentos na linguagem de programação Java; definir o conceito, a aplicação e a identificação de variáveis e constantes; demonstrar o uso dos operadores de atribuição, aritméticos, relacionais e lógicos tanto na notação algorítmica como na linguagem de programação Java; exemplificar a construção de expressões de atribuição, aritméticas e lógicas; mostrar a ordem de precedência matemática utilizada na resolução de problemas; apresentar a tabela-verdade como recurso que facilita o entendimento do uso dos operadores lógicos.

Os dados são, na verdade, os valores que serão utilizados para a resolução de um problema. Esses valores podem ser fornecidos pelo usuário do programa, podem ser originados a partir de processamentos (cálculos) ou, então, a partir de arquivos, bancos de dados ou outros programas.

NOTA:

Dados x informações – para alguns autores, os dados correspondem aos valores fornecidos na entrada e que serão processados, gerando uma informação.

Os dados são armazenados temporariamente em variáveis para que sejam processados de acordo com as especificações do algoritmo. Para que haja integridade no resultado obtido, os dados devem ser classificados de acordo com o tipo do valor a ser armazenado na variável, isto é, para evitar problemas que podem ser ocasionados devido ao fornecimento de valores inadequados à operação que será realizada. Por exemplo, vamos supor que o algoritmo deva especificar os passos para efetuar a soma de dois números quaisquer fornecidos pelo usuário. Então, será feita uma operação aritmética (soma), que só poderá ser realizada com valores numéricos.

Os tipos de dados são definidos, normalmente, a partir dos tipos primitivos criados em função das características dos computadores. Como os dados manipulados pelos computadores durante a execução dos programas são armazenados na memória, esses tipos seguem as características de formato e espaço disponível nessa memória. Assim, são organizados em bits e bytes e suas combinações. Por exemplo, para representar um número inteiro, poderiam ser usados dois bytes ou 16 bits. Isso resultaria em 2^{16} combinações possíveis para a representação de números, ou 65.536 possibilidades, considerando os estados possíveis que um bit pode assumir: 0 ou 1. Lembrando que os números poderiam assumir valores negativos e positivos nessa faixa, teríamos representações que iriam de -32.768 a 32.767, conforme pode ser verificado na tabela do item a seguir.

NOTA: *Byte – Conjunto de 8 bits que pode representar um caractere (letras, números ou símbolos especiais).*

Bit – A menor unidade de informação reconhecida pelo computador; pode representar os estados 0 (desligado) ou 1 (ligado).

3.1 TIPOS DE DADOS

Definir o tipo de dado mais adequado para ser armazenado em uma variável é uma questão de grande importância para garantir a resolução do problema. Ao desenvolver um algoritmo, é necessário que se tenha conhecimento prévio do tipo de informação (dado) que será utilizado para resolver o problema proposto. A partir daí, escolhe-se o tipo adequado para a variável que representará esse valor. Na confecção de algoritmos, utilizamos os tipos de dados primitivos (literal, inteiro, real e lógico), uma vez que os algoritmos apenas representam a resolução dos problemas. Já na confecção de programas, existem desdobramentos para esses tipos de dados a fim de adequá-los melhor ao propósito de cada linguagem e à resolução prática dos problemas. Veja na Tabela 1 as definições dos tipos de dados primitivos e seus desdobramentos na linguagem de programação Java.

LEMBRE-SE: *Alguns tipos de dados da linguagem de programação Java têm particularidades. Veja a seguir quais são elas.*

O tipo `long` deve ser identificado com a letra `L` para não ser ‘compactado’ pela linguagem em um tipo inteiro. A compactação ocorre como uma maneira de reduzir a memória gasta.

Da mesma forma que tenta usar o mínimo de memória, a linguagem Java tenta utilizar o máximo de precisão possível. Assim, se um elemento do tipo `float` (7 casas de precisão após a vírgula) não for identificado com a letra `f`, a linguagem vai considerá-lo como do tipo `double` (15 casas de precisão após a vírgula), o que pode gerar vários erros de compilação e execução.

Isso é uma característica da linguagem Java e não deve ser estendido a outras linguagens. Por exemplo:

```
int n = 4;
long numero = 456L;
float pi = 3.14f;
double tamanho = 3.873457486513793;
```

Primitivos		Específicos para linguagem de programação Java	
Tipos de dados	Definição	Tipos de dados	Capacidade de armazenamento na memória do computador, de acordo com a linguagem Java
Literal – também conhecido como texto ou caractere	Poderá receber letras, números e símbolos. <i>Obs.:</i> Os números armazenados em uma variável cujo tipo de dado é literal não poderão ser utilizados para cálculos.	char	16 bits – Armazena Unicodes. NOTA: Também é possível armazenar dados do tipo literal na Classe String.
Inteiro	Poderá receber números inteiros positivos ou negativos.	byte	8 bits – De (-128) até (127).
		short	16 bits – De (-32.768) até (32.767).
		int	32 bits – De (-2.147.483.648) até (2.147.483.647).
		long	64 bits – De (-9.223.372.036.854.775.808) até (9.223.372.036.854.775.807).
Real – também conhecido como ponto flutuante	Poderá receber números reais, isto é, com casas decimais, positivos ou negativos.	float	32 bits – De (-3,4E-38) até (-3,4E+38).
		double	64 bits – De (-1,7E-308) até (+1,7E+308).
Lógico – também conhecido como booleano	Poderá receber verdadeiro (1) ou falso (0).	boolean	8 bits – Em Java pode-se armazenar true ou false.

|TABELA 1 | Tabela de tipos de dados

CUIDADO!

Em Java, String é uma classe definida, não um tipo primitivo, mas é utilizado para armazenar cadeias de caracteres como o tipo de dado primitivo Literal.

3.1.1 TIPOS CONSTRUÍDOS

Nos algoritmos, assim como nas linguagens de programação, existe a possibilidade de criar outros tipos de dados, chamados **tipos construídos**. O tipo construído mais comum consiste na declaração de um conjunto de campos que compõe um registro. Por exemplo:

```
Algoritmo Exemplo_Registro
Tipo
    Reg_paciente = registro
        Nome: literal
        Idade: inteiro
        Peso: real
    fim_registro
Var
    Paciente: Reg_paciente
...
```

No **Exemplo_Registro**, o tipo **Reg_paciente** foi construído com um conjunto de campos (variáveis) de diversos tipos de dados primitivos. Após a construção, podem-se declarar variáveis que utilizem esse tipo. Em nosso exemplo, criamos a variável **Paciente**.

NOTA:

Em Java, um registro é uma class geralmente composta por vários campos.

3.2 VARIÁVEIS

Nos algoritmos, as variáveis são utilizadas para representar valores desconhecidos, porém necessários para a resolução de um problema e que poderão ser alterados de acordo com a situação. Por isso dizemos que as variáveis armazenam valores (dados) temporariamente.

Quando um algoritmo é transcrito para uma determinada linguagem de programação, as variáveis também terão a função de armazenar dados temporariamente, mas na memória RAM do computador. Esses dados serão utilizados durante o processamento para a resolução do problema em questão.

NOTA:

RAM (Random Access Memory) – Memória temporária para armazenamento dos programas que estão sendo executados no computador.

3.2.1 IDENTIFICAÇÃO DAS VARIÁVEIS PARA OS ALGORITMOS

Toda variável deve ser identificada, isto é, deve receber um nome ou identificador. O nome de uma variável deve ser único e deve estar de acordo com algumas regras:

- Não utilizar espaços entre as letras. Por exemplo, em vez de nome do cliente, o correto seria nome_do_cliente ou nomecliente. O caractere ‘sublinha’ ou ‘underline’ (_) pode ser utilizado para representar o espaço entre as letras.
- Não iniciar o nome da variável com algarismos (números). Por exemplo: não usar 2valor. O correto seria valor2.
- Não utilizar palavras reservadas, isto é, palavras que são utilizadas nos algoritmos para representar ações específicas. Por exemplo:
se palavra que representa uma condição ou teste lógico;
var palavra que representa a área de declaração de variáveis.
- Não utilizar caracteres especiais, como acentos, símbolos (? / : @# etc.), ç, entre outros.
- Ser sucinto e utilizar nomes coerentes.

LEMBRE-SE:

Cada linguagem de programação tem suas particularidades para declaração de variáveis. Essas particularidades devem ser conhecidas e observadas quando da atribuição dos nomes às variáveis.

3.2.2 IDENTIFICADORES DE VARIÁVEIS PARA A LINGUAGEM JAVA

Em Java, os nomes para as variáveis são *case-sensitive*, isto é, nomes com letras maiúsculas são diferenciados de nomes com letras minúsculas. Por exemplo: NomeCliente é diferente de nomecliente e também de nomeCliente.

- Nomes devem começar com uma letra, um caractere ‘sublinha’ ou ‘underline’ (_) ou o símbolo cifrão (\$). Os caracteres subsequentes podem também ser algarismos.
- Não utilizar caracteres especiais, como acentos, símbolos (? / : @# etc.), ç, entre outros, exceto os acima citados.
- As letras podem ser maiúsculas ou minúsculas.
- Não podem ser utilizadas palavras reservadas, como final, float, for, int etc.

3.3 CONSTANTES

São valores que não sofrem alterações ao longo do desenvolvimento do algoritmo ou da execução do programa. Por exemplo, na expressão abaixo, o valor 3.1415 é atribuído à constante pi e permanecerá fixo até o final da execução.

```
pi ← 3.1415;
perímetro ← 2 * pi * raio;
```

Em Java, uma constante é uma variável declarada com o modificador `final`. Por exemplo:

```
final float pi = 3.1415f;
```

NOTA: Modificadores são utilizados para modificar a atribuição de classes, variáveis ou métodos.

As constantes devem ser declaradas como variáveis cujo valor atribuído permanecerá inalterado ao longo do programa. Por isso, são também chamadas de **variáveis somente de leitura**.

3.4 OPERADORES

Os operadores são utilizados para representar expressões de cálculo, comparação, condição e atribuição. Temos os seguintes tipos de operadores: de atribuição, aritméticos, relacionais e lógicos.

3.4.1 OPERADORES DE ATRIBUIÇÃO

São utilizados para expressar o armazenamento de um valor em uma variável. Esse valor pode ser pré-definido (variante ou não) ou pode ser o resultado de um processamento.

Representação utilizando-se a notação algorítmica	Representação utilizando-se a notação para linguagem Java
←	=
Exemplo: <pre>nome ← "Fulano de tal" resultado ← a + 5 valor ← 3</pre>	Exemplo: <pre>nome = "Fulano de tal" resultado = a + 5 valor = 3</pre>

|TABELA 2| Operadores de atribuição simples

NOTA: Alguns autores utilizam := como sinal de atribuição em algoritmos.

3.4.2 OPERADORES ARITMÉTICOS

São utilizados para a realização dos diversos cálculos matemáticos. São eles:

Operador	Representação utilizando-se a notação algorítmica	Representação utilizando-se a notação para linguagem Java	Exemplos em Java
Incremento	Utiliza-se uma expressão. Por exemplo: $a+1$.	<code>++</code>	Adiciona 1 ao valor de a . Exemplo: <code>a++</code> – retorna o valor de a e depois adiciona 1 a esse valor; <code>++a</code> – adiciona 1 ao valor de a antes de retorná-lo.
Decremento	Utiliza-se uma expressão. Por exemplo: $a-1$.	<code>--</code>	Subtrai 1 do valor de a . Exemplo: <code>a--</code> – retorna o valor de a e depois subtrai 1 desse valor; <code>--a</code> – subtrai 1 do valor de a antes de retornar.
Multiplicação	<code>*</code>	<code>*</code>	$a * b$ – Multiplica a por b .
Divisão	<code>/</code>	<code>/</code>	a/b – Divide o valor de a por b .
Exponenciação	<code>^</code> ou <code>**</code> . Por exemplo: 2^3 é 2^3 .	Vide nota	
Módulo	<code>Mod.</code> Por exemplo: <code>a mod b</code> .	<code>%</code>	$a \% b$ – Retorna o resto da divisão inteira de a por b . Por exemplo, se o valor de a fosse 9 e o valor de b fosse 2, teríamos $9 \% 2$; o resultado da divisão seria 4 e o resto (mod) seria 1.
Adição	<code>+</code>	<code>+</code>	$a + b$ – O valor de a é somado ao valor de b .
Subtração	<code>-</code>	<code>-</code>	$a - b$ – Do valor de a é subtraído o valor de b .

|TABELA 3| Operadores aritméticos

NOTA:

Nem todos os operadores aritméticos utilizados na realização de cálculos podem ser diretamente representados por símbolos computacionais. Alguns deles são representados por funções matemáticas, como no caso da exponenciação e da radiciação. Em Java, essas operações e algumas outras são realizadas utilizando-se métodos da classe `Math`. Alguns desses métodos são mostrados no Anexo I.

Funções matemáticas são programas especiais existentes nas bibliotecas das linguagens de programação e executam cálculos matemáticos mais complexos não suportados pelos operadores matemáticos básicos.

3.4.3 OPERADORES RELACIONAIS

São utilizados para estabelecer uma relação de comparação entre valores ou expressões. O resultado dessa comparação é sempre um valor lógico (booleano) verdadeiro ou falso.

Operador	Representação utilizando-se a notação algorítmica	Representação utilizando-se a notação para linguagem Java	Exemplos em Java
Maior	>	>	$a > b$ – Se o valor de a for maior do que o valor de b , retornará verdadeiro. Senão, retornará falso.
Maior ou igual	\geq	\geq	$a \geq b$ – Se o valor de a for maior ou igual ao valor de b , retornará verdadeiro. Senão, retornará falso.
Menor	<	<	$a < b$ – Se o valor de a for menor do que o valor de b , retornará verdadeiro. Senão, retornará falso.
Menor ou igual	\leq	\leq	$a \leq b$ – Se o valor de a for menor ou igual ao valor de b , retornará verdadeiro. Senão, retornará falso.
Igual a	=	==	$a == b$ – Se o valor de a for igual ao valor de b , retornará verdadeiro. Senão, retornará falso.
Diferente de	\neq	!=	$a != b$ – Se o valor de a for diferente do valor de b , retornará verdadeiro. Senão, retornará falso.

|TABELA 4| Operadores relacionais

3.4.4 OPERADORES LÓGICOS

São utilizados para concatenar ou associar expressões que estabelecem uma relação de comparação entre valores. O resultado dessas expressões é sempre um valor lógico (booleano) verdadeiro ou falso.

3.4.5 PRECEDÊNCIA DOS OPERADORES

As linguagens de programação normalmente estabelecem uma ordem de avaliação considerando a precedência dos operadores quando mais de um operador é usado em uma expressão. Consulte o anexo para obter a precedência dos operadores para a linguagem Java. No caso de não haver precedência entre os operadores aplicados, a expressão é avaliada da esquerda para a direita.

Considere a expressão: $A = B < 8$. e. $C = 3$. Essa expressão combina duas expressões de comparação: $B < 8$ e $C = 3$. Conforme visto, se ambas resultarem em

Operador	Representação utilizando-se a notação algorítmica	Representação utilizando-se a notação para linguagem Java	Exemplos em Java
e	.e.	&&	<code>a = 5 && b != 9</code> – Caso o valor de a seja igual a 5 e o valor de b seja diferente de 9, então retornará verdadeiro. Caso contrário, retornará falso.
ou	.ou.		<code>a = 5 b != 9</code> – Caso o valor de a seja igual a 5 ou o valor de b seja diferente de 9, então retornará verdadeiro. Se ambas as comparações retornarem falso, o resultado será falso.
não	.não.	!	<code>! a > 5</code> – Se o valor de a for maior do que 5, retornará falso. Caso contrário, retornará verdadeiro.

|TABELA 5| Operadores lógicos

verdadeiro, o valor verdadeiro será atribuído à variável A. Em qualquer outra circunstância, o valor falso será atribuído a A. Deve-se observar que, primeiramente, são avaliadas as expressões de comparação B < 8 e C = 3 e, posteriormente, os resultados dessas duas expressões são associados por meio do operador .e., obtendo-se o resultado final. Isso se deve ao fato de existir uma precedência entre os operadores relacionais e os lógicos. Os operadores relacionais são avaliados primeiro e, posteriormente, os lógicos. Normalmente, além da existência de precedência entre operadores de tipos diferentes, operadores do mesmo tipo possuem, também, uma precedência. A Tabela 6 demonstra essa relação.

Operador	Observação
(), []	Parênteses e colchetes são usados para agrupar expressões, determinando precedência, a exemplo das expressões matemáticas.
^ ou **	Operador aritmético de potenciação.
*, /	Operadores aritméticos de multiplicação e divisão.
+, -	Operadores aritméticos de adição e subtração.
←	Operador de atribuição.
=, <, >, <=, >=, <>	Operadores relacionais.
.não.	Operador lógico de negação.
.e.	Operador lógico e.
.ou.	Operador lógico ou.

|TABELA 6| Precedência de operadores

Dessa forma, a expressão:

$A \leftarrow B + 2 > 5 \text{ .ou. } C \leftrightarrow 4 \text{ .e. } D = 0$

seria avaliada na seguinte ordem:

- $C \leftrightarrow 4$ [1]
- $D = 0$ [2]
- [1] .e. [2] [3]
- $B + 2$ [4]
- [4] > 5 [5]
- [5] .ou. [3] [6]
- $A \leftarrow [6]$

LEMBRE-SE: A precedência matemática também deve ser considerada na implementação dos algoritmos.

3.5 TABELA-VERDADE

A **tabela-verdade** expressa o conjunto de possibilidades existentes para a combinação de variáveis ou expressões e operadores lógicos. Um exemplo de combinação entre variáveis é $A \geq 5 \text{ .e. } B \neq 10$, onde A e B são as variáveis, \geq e \neq são os operadores relacionais e .e. é o operador lógico. Um exemplo de combinação entre expressões é $A + B \neq X - 10$. A tabela-verdade é utilizada para facilitar a análise da combinação dessas expressões ou variáveis, conforme pode ser verificado a seguir:

			Operador		
			$\&\& \text{ (.e.)}$	$\ \text{ (.ou.)}$	$! \text{ (.não.)}$
Expressão algoritmo	$A = 5$	$B \neq 9$	$A = 5 \text{ .e. } B \neq 9$	$A = 5 \text{ .ou. } B \neq 9$	$.\text{não. } A = 5$
Expressão em Java	$A == 5$	$B != 9$	$A == 5 \&& B != 9$	$A == 5 B != 9$	$!\text{A == 5}$
Resultados possíveis	.v.	.v.	.v.	.v.	.f.
	.v.	.f.	.f.	.v.	.f.
	.f.	.v.	.f.	.v.	.v.
	.f.	.f.	.f.	.f.	.v.

|TABELA 7| Tabela-verdade

Na tabela, pode-se verificar que as expressões $A = 5$ e $B \neq 9$ podem assumir quatro possibilidades. Ou seja, ambas podem ser verdadeiras (primeira linha dos resultados possíveis), a primeira pode ser verdadeira e a segunda falsa, a primeira pode ser falsa e a segunda verdadeira ou ambas podem ser falsas. Essas combinações dependem, portanto, dos valores atribuídos às variáveis A e B.

Submetendo essas expressões aos operadores lógicos (`&&` – e, `||` – ou, `!` – não), obtém-se valores diferentes, dependendo do resultado que cada uma das expressões assumir individualmente. Assim, considerando a primeira linha de resultados possíveis, onde $A = 5$ é verdadeiro e $B <> 9$ também é, obtemos os seguintes resultados:

- verdadeiro para $A = 5$.e. $B <> 9$: se $A = 5$ é verdadeiro e $B <> 9$ também é, o resultado associado com o operador `&&` também é;
- verdadeiro para $A = 5$.ou. $B <> 9$: se $A = 5$ é verdadeiro e $B <> 9$ também é, o resultado associado com o operador `||` também é;
- falso para .não. $A = 5$: se $A = 5$ é verdadeiro, a negação é falso.

Deduz-se que, para o operador `&&`, o resultado será verdadeiro somente se ambas as expressões associadas assumirem o resultado verdadeiro. Por outro lado, para o operador `||`, o resultado será verdadeiro se pelo menos uma das expressões associadas assumir o resultado verdadeiro.

3.6 EXERCÍCIOS PARA FIXAÇÃO

1. Dadas as expressões a seguir, identificar o resultado verdadeiro ou falso que cada uma delas retornaria, em função dos valores dados.

Exemplo:

Supondo que à variável A seja atribuído o valor 2 e à variável B seja atribuído o valor 7:

$$A = 2 \text{ .e. } B = 5$$

Resultado: falso (para $A=2$, o resultado é verdadeiro; para $B=5$, o resultado é falso. Como o operador é .e., o resultado final é falso).

Considerando os mesmos valores atribuídos a A e B , avalie as expressões a seguir:

- a) $A = 3$.e. $B = 7$
- b) $A < 3$.ou. $B <> 7$
- c) $A \leq 2$.e. $B = 7$
- d) .não. $A = 2$.e. $B = 7$
- e) $A < 5$.e. $B > 2$.ou. $B <> 7$

2. Verifique se as variáveis abaixo possuem nomes corretos e justifique as alternativas falsas:

- | | | |
|-----------------------|-----------------------|-----------------------|
| a) <code>n#1</code> | b) <code>tempo</code> | c) <code>n_1</code> |
| d) <code>\$din</code> | e) <code>n 1</code> | f) <code>K2K</code> |
| g) <code>n1</code> | h) <code>U F</code> | i) <code>2nome</code> |

j) dep

k) nome2

l) val#r

3. Sabe-se que o uso **incorrecto** da precedência de operadores ocasiona erros. Pensando nisso, avalie as expressões a seguir e:

- a) classifique a ordem em que as operações deverão ser executadas;
 b) determine o resultado das operações.

Considere os seguintes valores para as variáveis:

$$A \leftarrow 8; B \leftarrow 5; C \leftarrow -4; D \leftarrow 2$$

- a) Delta $\leftarrow B^2 - 4 * A * C$
- b) J $\leftarrow "Hoje" <> "HOJE"$
- c) Media $\leftarrow (A + B + C + D) / 4$
- d) Media $\leftarrow A + B + C + D / 4$
- e) Resultado $\leftarrow A \bmod D / 5$
- f) Resultado $\leftarrow (A \bmod D) / 5$
- g) X $\leftarrow (A + B) - 10 * C$
- h) X $\leftarrow A + B - 10 * C$
- i) Y $\leftarrow A > 8 .e. B + C > D$
- j) Y $\leftarrow A > 3 * 2 .ou. B + C <> D$

3.7 EXERCÍCIOS COMPLEMENTARES

1. Considere a seguinte atribuição de valores para as variáveis:

$$A \leftarrow 3, B \leftarrow 4 \text{ e } C \leftarrow 8$$

Avalie as expressões a seguir indicando o resultado final: verdadeiro ou falso.

- a) $A > 3 .e. C = 8$
- b) $A <> 2 .ou. B \leq 5$
- c) $A = 3 .ou. B \geq 2 .e. C = 8$
- d) $A = 3 .e. \text{não. } B \leq 4 .e. C = 8$
- e) $A <> 8 .ou. B = 4 .e. C > 2$
- f) $B > A .e. C <> A$
- g) $A > B .ou. B < 5$
- h) $A <> B .e. B = C$
- i) $C > 2 .ou. A < B$
- j) $A > B .ou. B > A .e. C <> B$

2. Complete a tabela-verdade a seguir:

			Operador		
			.e.	.ou.	.não.
Expressão	A = 4	B <> 5	A = 4 .e. B <> 5	A = 4 .ou. B <> 5	.não. A = 4
Resultados possíveis					

3. Construa a tabela-verdade para as expressões:

- a) A \geq 3 .ou. B = 5
- b) A \neq 9 .e. B \leq 6
- c) .não. A = 2 .ou. B \geq 1
- d) A $>$ 3 .e. B \neq 5 .ou. C $<$ 8

4. Dada a declaração de variáveis:

```
Var A,B,C: inteiro;
          X,Y,Z: real;
          Nome, Rua: literal;
          L1: lógico;
```

e atribuindo-se a essas variáveis os valores:

A \leftarrow 1	X \leftarrow 2,5	Nome \leftarrow "Pedro"
B \leftarrow 2	Y \leftarrow 10,0	Rua \leftarrow "Girassol"
C \leftarrow 3	Z \leftarrow -1,0	L1 \leftarrow .v.

Determine o resultado das expressões a seguir:

- a) Nome = Rua
- b) X $>$ Y .e. C \leq B
- c) (C - 3 * A) $<$ (X + 2 * Z)
- d) ((Y / 2) = X) .ou. ((B * 2) \geq (A + C))
- e) .não. L1
- f) .não. C = B .e. X + Y \leq 20 .ou. L1 \neq .v.

CONCEITOS DE PROGRAMAÇÃO

- ▶ *Noções gerais sobre as diferentes formas de programação*
- ▶ *Características e vantagens das linguagens de programação*
- ▶ *Aplicabilidade das diferentes linguagens*
- ▶ *Exercícios para fixação*
- ▶ *Exercícios complementares*



OBJETIVOS:

Abordar os principais paradigmas em programação para que o leitor passe a conhecer os diferentes tipos de linguagens de programação e suas aplicações, pois o pseudocódigo é uma forma estruturada de representação das soluções de problemas e a linguagem de programação Java, a qual será utilizada para codificar os exemplos, é orientada a objetos; apresentar os principais conceitos de programação orientada a objetos.

Um programa é um conjunto de instruções que dizem ao computador o que deve ser feito. Existem muitas formas e diferentes tipos de linguagens de programação, cada qual com uma finalidade específica. Pode-se até dizer que as linguagens de programação podem ser classificadas em níveis, segundo sua finalidade, muitos deles coexistentes em um mesmo computador.

Os primeiros computadores, como o Eniac e o Univac, consistiam principalmente em válvulas e relés e tinham de ser programados conectando-se uma série de plugues e fios. Uma equipe de programadores podia passar dias introduzindo um pequeno programa em uma dessas máquinas, que ocupavam salas inteiras.

É aí que entram os diversos níveis de programação. Hoje em dia, existem linguagens de programação que atuam diretamente no hardware da máquina, movimentando dados e acionando dispositivos ligados ao computador. É a chamada **linguagem de baixo nível**. Como esse tipo de linguagem é de difícil programação, a exemplo do que ocorria com os primeiros computadores, criaram-se níveis de linguagens. Assim, as **linguagens de alto nível** como Pascal, C, C++ e Java são utilizadas pelos programadores no desenvolvimento de programas. Os programas digitados nessas linguagens constituem o que se chama de **código-fonte**, o qual é convertido (traduzido) para programas de baixo nível, em um processo chamado de **compilação** ou **interpretação**. O programa assim traduzido pode ser executado pela máquina, reproduzindo aquilo que o programador deseja.

Embora as linguagens de programação consideradas de alto nível tenham o objetivo de aproximar-se da linguagem humana, esse objetivo ainda está longe de ser alcançado. O máximo que se conseguiu foi criar instruções mnemônicas, em inglês, para facilitar o processo.

NOTA:

*Eniac (Electronic Numerical Integrator and Calculator) – é considerado o primeiro computador digital eletrônico.
Univac – primeiro computador a ser comercializado.*

4.1 INTRODUÇÃO À PROGRAMAÇÃO

Apesar de existirem vários níveis de programação, neste livro serão tratadas as linguagens de alto nível, utilizadas para o desenvolvimento dos algoritmos, objeto de nosso estudo.

A linguagem de programação, como qualquer linguagem, é formada por palavras. Essas palavras são agrupadas em frases para produzir um determinado significado. Dessa forma, pode-se chamar as palavras de uma linguagem de programação de **palavras-chave** e as frases criadas com essas palavras de **estruturas de programação**.

Assim, um programa é constituído de palavras-chave e estruturas de programação definidas segundo as regras dessa linguagem, elaboradas de modo que sejam mais facilmente compreendidas pelo ser humano. A exemplo da linguagem usada em nossa comunicação no dia-a-dia, a linguagem de programação possui uma sintaxe, definida por essas regras.

Por que existem tantos tipos de linguagem? Uma linguagem é melhor que outra? A resposta para essas perguntas está justamente no objetivo para o qual elas foram criadas. Cada linguagem de programação foi desenvolvida para solucionar determinado tipo de problema e cumprir uma função determinada. Uma linguagem pode ser melhor para a execução de cálculos matemáticos complexos, com aplicações na área científica; outra pode ser melhor para processar uma grande quantidade de dados submetidos a operações simples, com aplicações na área financeira; e ainda outras exigem uma interface

elaborada e fácil interação com o usuário. Como se pode ver, a linguagem que você escolhe para o desenvolvimento de uma aplicação depende da sua adequação à tarefa que se pretende executar. Como exemplo, podem-se citar algumas das linguagens de programação mais comumente utilizadas:

PASCAL

É uma linguagem de alto nível poderosa e eficientemente estruturada. Criada para ser uma ferramenta educacional pela simplicidade de sua sintaxe, vem sendo utilizada até hoje nos meios acadêmicos.

A linha de código abaixo exemplifica uma instrução em Pascal para exibir uma frase na tela do computador.

```
WRITE ("Algoritmos e Estruturas de Dados");
```

C

Linguagem estruturada utilizada até pouco tempo para o desenvolvimento de aplicações comerciais. Ultimamente, tem grande aplicação no desenvolvimento de software básico e aplicações com forte interação com o hardware. A mesma instrução do exemplo anterior pode ser escrita em C como:

```
printf("Algoritmos e Estruturas de Dados");
```

C++

Linguagem de alto nível orientada a objetos; uma evolução do C que preserva seus princípios de eficiência e facilidade de programação. A programação orientada a objetos será discutida ainda neste capítulo. O exemplo anterior pode ser reproduzido com a instrução:

```
cout<<"Algoritmos e Estruturas de Dados";
```

JAVA

Linguagem orientada a objetos de fácil programação e larga utilização no mercado. Amplamente utilizada em aplicações de processamento distribuído e para a Internet. Java está sendo utilizada neste livro para exemplificar a implementação dos algoritmos estudados. Uma descrição mais detalhada da linguagem encontra-se no Anexo. Reproduzindo o exemplo anterior, tem-se:

```
System.out.println("Algoritmos e Estruturas de Dados");
```

Como podemos observar, existe grande semelhança na sintaxe utilizada pelas diversas linguagens de programação e o aprendizado de uma delas depende de convívio maior e utilização freqüente. Um bom programa é aquele que tem, dentre outras qualidades, um código eficiente. Programas eficientes são desenvolvidos com técnicas de programação adequadas e algoritmos eficientemente projetados.

4.2 TIPOS DE PROGRAMAÇÃO

Existem formas diferentes de se programar, mesmo utilizando a mesma linguagem de programação. É como se fossem comparados textos que tratam do mesmo assunto mas que foram escritos por pessoas diferentes.

Apesar disso, as linguagens de programação possuem características e regras que determinam como o programa deverá ser escrito para que seja ‘interpretado’ pela máquina.

4.2.1 CONCEITOS SOBRE A PROGRAMAÇÃO LINEAR

A **programação linear** pressupõe a criação de programas que, na sua execução, obedeçam a uma seqüência de passos executados consecutivamente, com início e fim específicos. Esse princípio era utilizado por linguagens mais antigas, como o Basic, no qual as linhas de código eram numeradas uma a uma e eventuais desvios eram executados apontando-se para a linha desejada.

Embora alguns tipos de linguagem tenham restrições na forma de programação, dando pouca flexibilidade ao desenvolvedor, nada impede que programas lineares sejam gerados utilizando-se de linguagens estruturadas ou orientadas a objetos.

A desvantagem da programação linear é a complexidade. Programas lineares extensos são difíceis de ser desenvolvidos e até compreendidos.

4.2.2 CONCEITOS SOBRE A PROGRAMAÇÃO ESTRUTURADA

Usando o velho provérbio “dividir para conquistar”, pode-se afirmar que, para a consecução de um objetivo, é melhor e bem mais fácil dividir as tarefas a serem realizadas em etapas, executando-as uma por vez, até que todo o trabalho tenha sido realizado. Pode-se, ainda, pensar que esse trabalho pode ser realizado em equipe. A divisão do trabalho e a distribuição adequada das tarefas a cada um dos elementos da equipe, se bem coordenadas, com certeza produzirão resultados melhores e bem mais rápidos. O problema, muitas vezes, está em como dividir essas tarefas de forma adequada e equitativa e gerenciar o seu desenvolvimento.

Em programação, pode-se pensar da mesma forma. A construção de um programa monolítico que execute uma tarefa complexa, além de ser complicada e demorada, pode não produzir o resultado esperado. Mesmo que o resultado seja alcançado, o tempo despendido pode ser demasiado. Essa situação se agrava se for considerado que esse tempo gasto representa custos no desenvolvimento de uma aplicação que podem superar as expectativas de custos do projeto.

Deve-se utilizar esta filosofia para criar um programa extenso e complicado. Um bom programador deve ser como um bom gerente no momento de atribuir tarefas em um programa: deve separar as tarefas que o programa deve realizar, depois deve atacá-las uma a uma, tornando o trabalho menos assustador. Essa abordagem é o que norteia o conceito de **programação estruturada**.

Na programação estruturada, a divisão de tarefas é um processo chamado de **modularização**. Nesse processo, divide-se o programa em partes ou módulos que executam tarefas específicas. É importante que essas tarefas, representadas pelos módulos, sejam específicas e cada uma delas seja o mais independente possível das demais realizadas por outros módulos do programa. Isto é, a independência funcional está relacionada diretamente à modularização. Por exemplo, suponha que você pertença a uma equipe que está participando de uma gincana na qual várias tarefas são determinadas e o prazo final está estabelecido. Vence a equipe que completar o maior número de tarefas dentro do prazo. Se as tarefas forem divididas adequadamente, será grande a chance de que todas sejam cumpridas no tempo determinado. Por outro lado, se ocorrer um erro nessa distribuição e alguém acabar fazendo algo que outro elemento do grupo também esteja fazendo, com certeza o tempo não será suficiente ou alguma tarefa deixará de ser realizada.

Os **procedimentos** ou **funções** são blocos de programa que executam determinada tarefa. Cada um desses blocos de código recebe um nome, o qual é utilizado como chamada do procedimento ou função. Tanto procedimentos quanto funções podem receber valores para que possam realizar suas tarefas. A diferença é que os procedimentos, embora possam receber valores, não retornam outros valores como resultado, enquanto uma função retorna valores como resultado das operações que realizou.

Os nomes dos procedimentos, normalmente, são palavras ou pequenas frases que procuram associá-los de forma mnemônica à tarefa realizada. Por exemplo:

AtualizarDados()

As funções são nomeadas da mesma maneira que os procedimentos, lembrando que o nome de uma função pode ser utilizado em uma expressão como se fosse uma variável, porque uma função retorna um valor quando termina sua execução. Por exemplo:

soma(x, y).

A seguir é apresentado um pequeno programa em C, que executa a soma de dois números introduzidos pelo teclado e exibe o resultado no vídeo.

```
1. /* funcao.c */
2. main()
3. {
4.     int x, y, r;
5.     printf("Digite dois números: ");
6.     scanf("%d %d", &x, &y);
7.     r = soma(x, y);
8.     printf("A soma dos números é: %d", r);
9. }
10. /* soma() */
11. /* retorna a soma de dois numeros */
12. soma(j, k)
```

```

13. int j, k;
14. {
15.     return(j+k);
16. }

```

No exemplo, as variáveis `x` e `y` recebem os números cuja entrada é feita pelo teclado (linha 6). A função `soma()` é chamada na linha 7, com a passagem dos valores. A função é executada a partir da linha 12 e, na linha 15, retorna o valor da soma dos números.

O exemplo é bastante simples e, obviamente, não seria necessário criar uma função apenas para somar dois números. Serve apenas como ilustração para a chamada de uma função com passagem e retorno de valores.

Ainda dentro do exemplo, podem-se encontrar outras funções internas da própria linguagem, como `printf()` e `scanf()` que funcionam, a princípio, da mesma forma que a função `soma()`.

Estudaremos outros exemplos de procedimentos e funções no Capítulo 7, que tratará com mais profundidade desse assunto.

Quando se pensa na criação de um programa, surge a pergunta: por onde devo começar? Uma forma de desenvolver um programa é partir de sua representação em pseudocódigo ou fluxograma, conforme discutido no Capítulo 2. O fluxograma é um tipo de algoritmo que utiliza símbolos gráficos para representar as ações ou instruções a serem seguidas. Assim como o pseudocódigo, ele é utilizado para organizar o raciocínio lógico a ser seguido para a resolução de um problema ou para a definição dos passos na execução de uma tarefa.

Cada etapa importante definida em seu fluxograma ou pseudocódigo pode ser usada como um nome de sub-rotina (procedimento ou função) no programa. Essas etapas devem estar definidas e claramente identificadas em seu esboço, de forma que se possa saber onde são utilizadas.

4.3 CONCEITOS SOBRE A PROGRAMAÇÃO ORIENTADA A OBJETOS

No item anterior, foram vistos os conceitos de programação estruturada e de como se pode, partindo de um programa extenso e complexo, subdividi-lo em procedimentos ou funções. Essa subdivisão auxilia no desenvolvimento, de forma que módulos pequenos e específicos ficam mais fáceis de ser programados e compreendidos. Esses módulos executam tarefas determinadas, interagindo com outros módulos ou com o programa principal, retornando valores ou não, dependendo do que deve ser realizado.

A **programação orientada a objetos** representa uma mudança no enfoque da programação, na forma como os sistemas eram vistos até então. Representa uma quebra de paradigma, revolucionando todos os conceitos de projeto e desenvolvimento de sistemas existentes anteriormente.

NOTA:

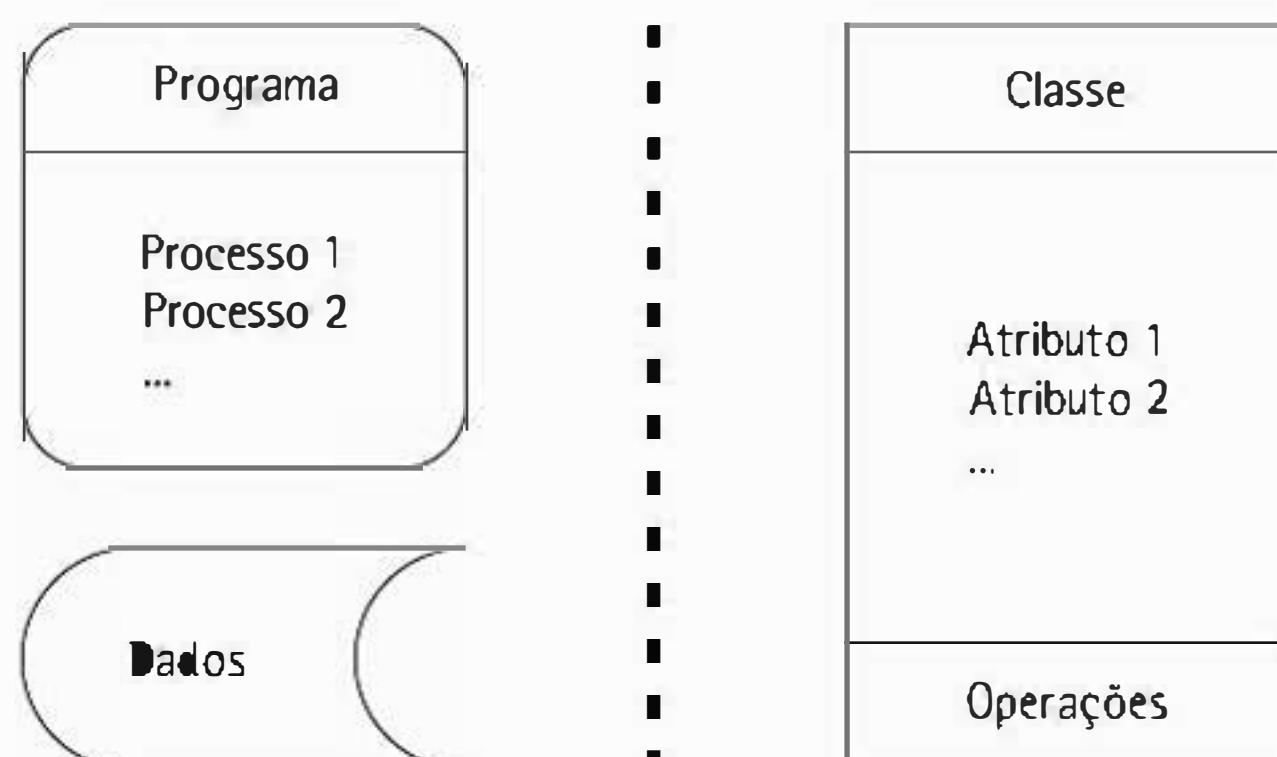
"Paradigma é um conjunto de regras que estabelecem fronteiras e descrevem como resolver os problemas dentro dessas fronteiras. Os paradigmas influenciam nossa percepção; ajudam-nos a organizar e a coordenar a maneira como olhamos para o mundo..."

Daniel C. Morris e Joel S. Brandon,
Reengenharia: reestruturando sua empresa,
São Paulo, Makron Books, 1994.

O enfoque tradicional para o desenvolvimento de sistemas e, por consequência, para a programação, baseia-se no conceito de que um sistema é um conjunto de programas inter-relacionados que atuam sobre um determinado conjunto de dados que se deseja manipular de alguma forma para obter os resultados desejados. O enfoque da modelagem de sistemas por objetos procura enxergar o mundo como um conjunto de objetos que interagem entre si e apresentam características e comportamento próprios representados por seus atributos e suas operações. Os atributos estão relacionados aos dados, e as operações, aos processos que um objeto executa.

Assim, supondo que se deseje desenvolver um sistema de controle de estoque para uma empresa, procura-se identificar os objetos relacionados ao sistema, como os produtos, os pedidos de compra, os recibos, as pessoas etc., conforme está detalhado a seguir. Pode-se dizer que é possível modelar, por meio da orientação a objetos, um setor, um departamento e até uma empresa inteira.

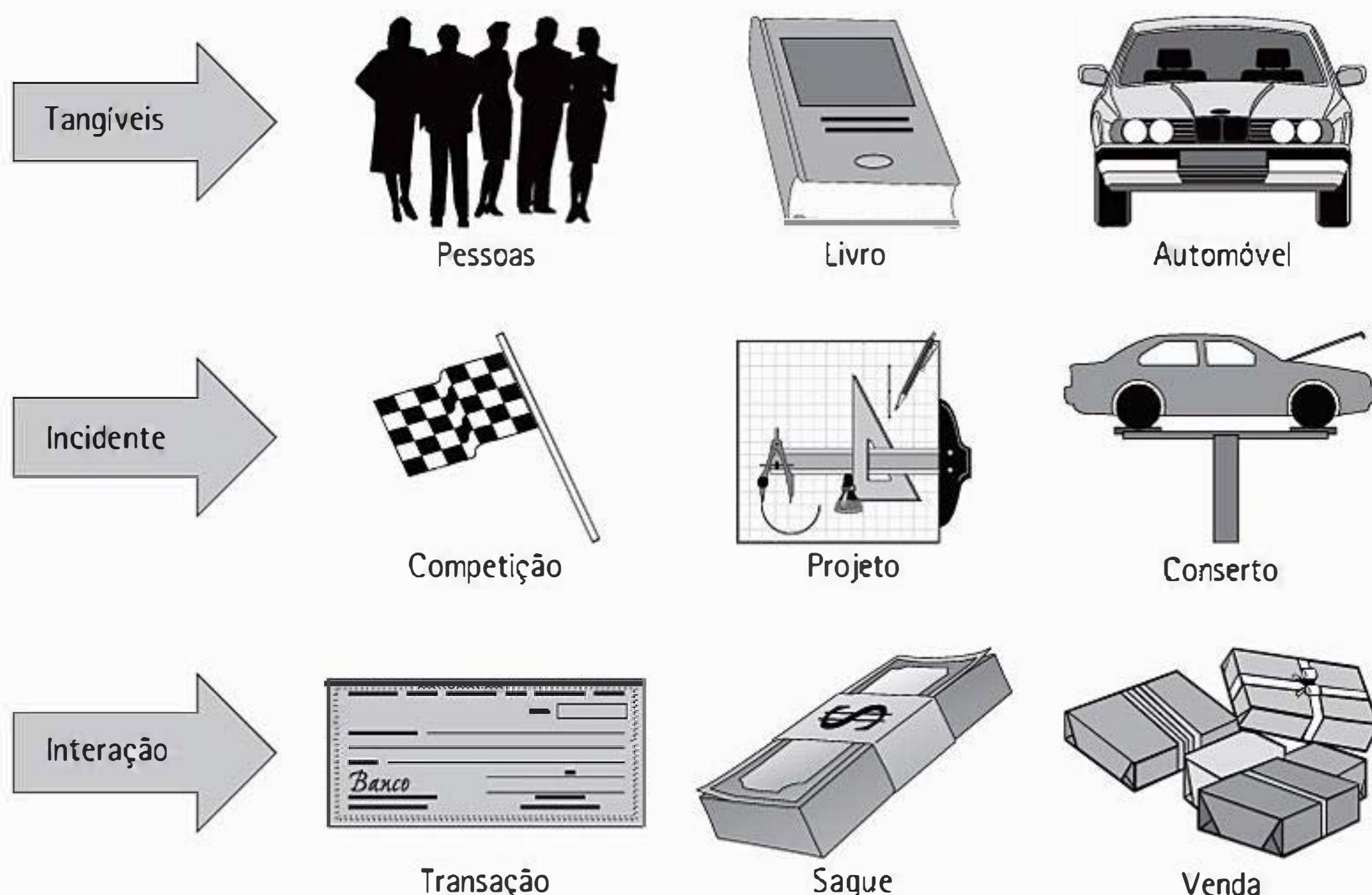
Esse enfoque justifica-se, de forma resumida, pelo fato de que os objetos existem na natureza muito antes de haver qualquer tipo de negócio envolvido ou qualquer tipo de sistema para controlá-los. Equipamentos, pessoas, materiais, produtos, peças, ferramentas, combustíveis etc. existem por si sós e possuem características próprias determinadas pelos seus atributos (nome, tamanho, cor, peso) e um determinado comportamento no mundo real relacionado aos processos que eles sofrem.



|FIGURA 1 | Enfoque tradicional x enfoque orientado a objetos

4.3.1 O QUE É UM OBJETO

Um dos primeiros conceitos básicos da orientação a objetos é o do próprio objeto. Um **objeto** é uma extensão do conceito de objeto do mundo real, em que se podem ter coisas *tangíveis*, um *incidente* (evento ou ocorrência) ou uma *interação* (transação ou contrato).



| FIGURA 2 | Tipos de objetos

Por exemplo, em um sistema acadêmico em que João é um *aluno objeto* e Carlos é um *professor objeto* que ministra *aulas objeto* da *disciplina objeto* algoritmos, para que João possa assistir às aulas da disciplina do prof. Carlos, ele precisa fazer uma *matrícula objeto* no *curso objeto* de computação.

Têm-se as ocorrências de objetos citados:

- tangíveis: aluno e professor;
- incidente: curso, disciplina, aula;
- interação: matrícula.

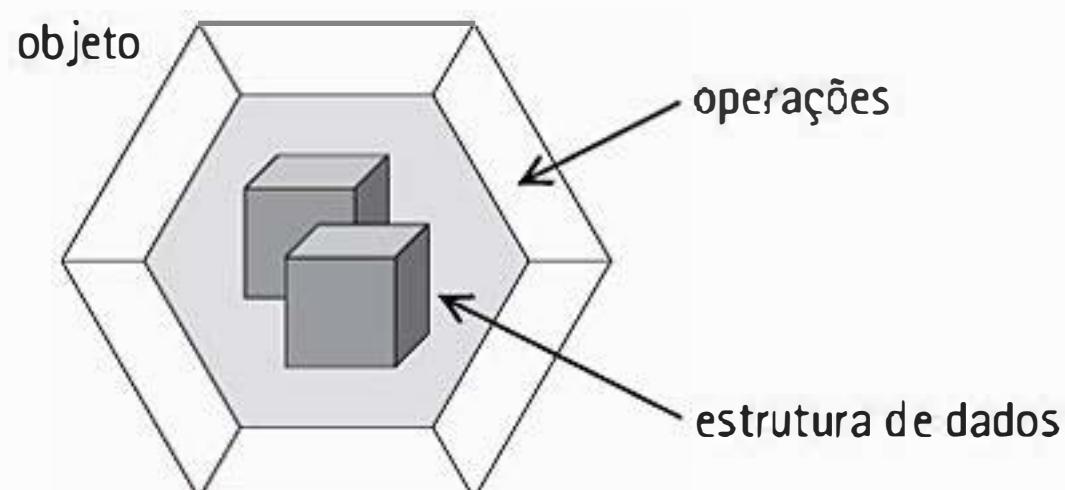
A identificação dos objetos em um sistema depende do nível de abstração de quem faz a modelagem, podendo ocorrer a identificação de diferentes tipos de objetos e diferentes tipos de classificação desses objetos. Não existe um modelo definitivamente correto; isso vai depender de quem faz a modelagem e de processos sucessivos de refinamento, até que se possa encontrar um modelo adequado a sua aplicação.

NOTA:

Abstração consiste em se concentrar nos aspectos essenciais, próprios, de uma entidade e em ignorar suas propriedades acidentais. Isso significa concentrar-se no que um objeto é e faz, antes de decidir como ele deve ser implementado em uma linguagem de programação.

4.3.2 COMO VISUALIZAR UM OBJETO?

Pode-se imaginar um objeto como algo que guarda dentro de si os dados ou informações sobre sua estrutura (seus atributos) e possui um comportamento definido pelas suas operações.



|FIGURA 3| Visualização do objeto

Os dados ficam protegidos pela interface, que se comunica com os demais objetos do sistema. Nessa interface, representada pela camada mais externa de nosso modelo, estão as operações. Todo tipo de alteração nos dados do objeto (atributos) somente poderá ser feito por meio das operações, que recebem as solicitações externas, fazem as alterações nos dados (se permitidas) e retornam outras informações para o meio externo.

4.3.3 CONCEITO DE CLASSES

O conceito de classes é muito importante para o entendimento da orientação a objetos.

Uma classe é uma coleção de objetos que podem ser descritos por um conjunto básico de atributos e possuem operações semelhantes. Falamos em um conjunto básico de atributos e operações semelhantes, pois veremos adiante que nem todos os objetos da mesma classe precisam ter exatamente o mesmo conjunto de atributos e operações.

Quando um objeto é identificado com atributos e operações semelhantes em nosso sistema, diz-se que pode ser agrupado em uma classe. Esse processo é chamado de **generalização**. Por outro lado, pode ocorrer que um objeto, ao ser identificado, constitua-se, na verdade, de uma classe de objetos, visto que dele podem se derivar outros objetos. Já esse processo é chamado de **especialização**.

Na Figura 4, partindo-se de uma classe *veículos*, observa-se a existência de vários tipos de veículos e a criação de especializações como: os *utilitários*, os veículos *esporte*, os de *passeio* e os de transporte de *passageiros*. Porém, se os veículos *utilitários*, *esporte*, de *passeio* e de transporte de *passageiros* são considerados tipos de veículos, pode-se generalizar criando-se a classe *veículos*.

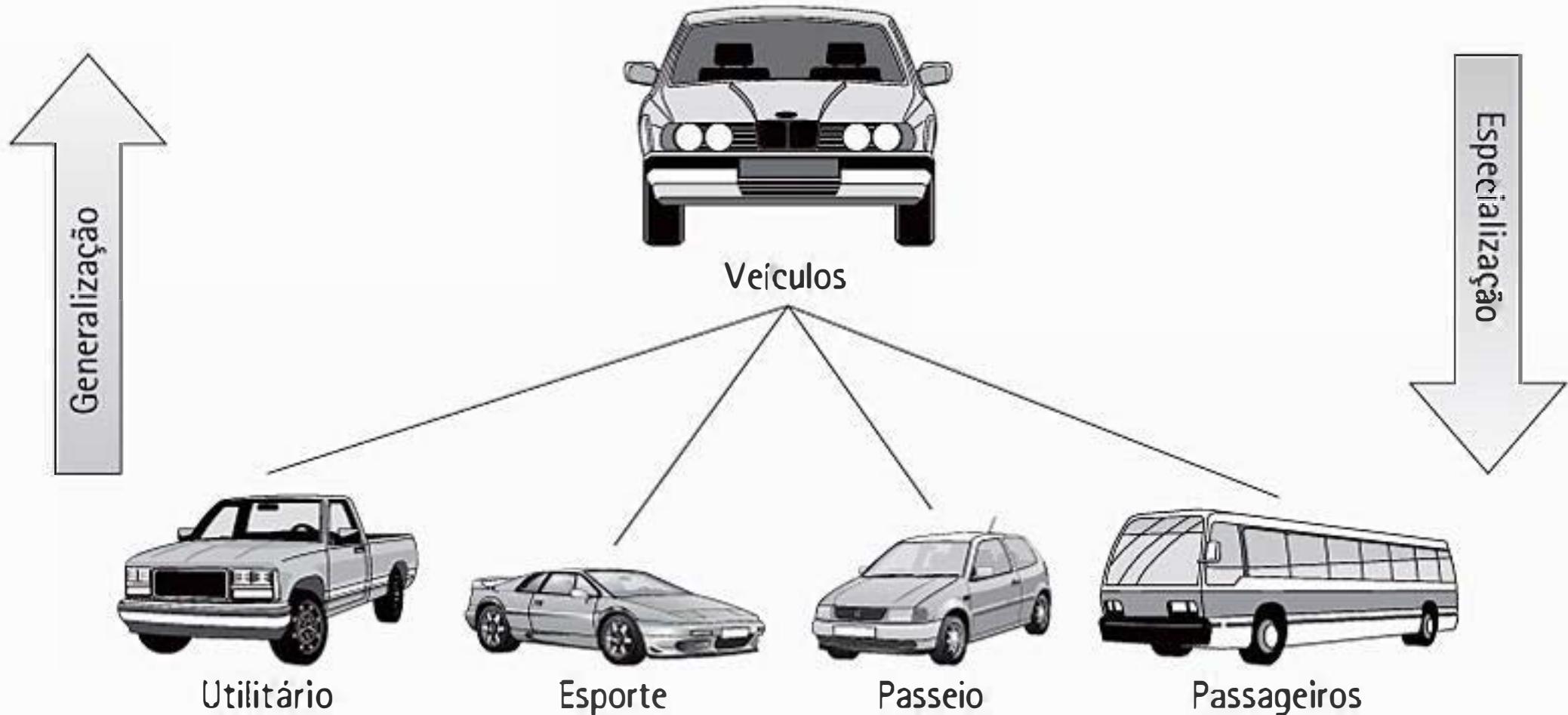


FIGURA 4 | Generalizaçāo e especializaçāo

O critério para criar a classe ou as especializações dessa classe está relacionado aos atributos e operações de cada um dos objetos. Assim, pode-se dizer que, para todos os tipos de veículos, têm-se atributos genéricos como: *marca*, *modelo*, *ano* de fabricação, *potência* do motor, número de *eixos*, capacidade de *carga* etc. Têm-se também operações semelhantes para todos como: *dar partida*, *acelerar*, *frear*, *estacionar* etc.

Nada impede, contudo, que os veículos de *passeio* sejam subdivididos em outros tipos, como os tipos *sedan*, *minivan* etc. Nesse caso, a classe *veículos* teria uma subclasse *passeio* e dessa derivariam os veículos *sedan* e as *minivans*.

4.3.4 INSTÂNCIAS DE OBJETOS

Quando se fala em classes de objetos, está sendo considerado que se podem incluir objetos em cada uma delas. Como exemplo, considere que será modelado um sistema para uma revendedora de veículos que comercializa os veículos conforme o esquema citado. Cada novo veículo adquirido pela revendedora seria cadastrado no sistema obedecendo a sua classificação. Supondo que o veículo seja um automóvel de passeio do tipo *sedan*, cria-se um novo objeto dessa classe, que será chamada de uma **instância** de objeto, conforme o seguinte esquema:

Classe Veículos	Subclasse Passeio	Subclasse Sedan	Instância marca: Opel modelo: Fire ano: 2002 potência: 195cv eixos: 2 carga: 1.500kg.	Instância marca: Thunderbird modelo: Hatch ano: 2000 potência: 250cv eixos: 2 carga: 1.800kg.
--------------------	----------------------	--------------------	---	---

Dessa forma, para cada novo veículo adquirido seria criada uma nova instância de objeto de uma determinada classe.

4.3.5 HERANÇA

Foi dito anteriormente que uma classe é constituída de objetos com atributos e operações semelhantes. Esse princípio orienta a implementação da **herança**. A herança nada mais é do que a implementação da generalização; é o compartilhamento de atributos e operações entre classes com base em um relacionamento hierárquico. Quando se cria uma nova instância de um objeto, dizemos, em orientação a objetos, que esse novo objeto *herda* os atributos e operações de sua classe.

No exemplo de instância de objeto, foi visto que um novo veículo adquirido possuía os seguintes atributos: marca: Opel, modelo: Fire, ano: 2002, potência: 195cv, eixos: 2 e carga: 1.500kg. Esses atributos são herdados da classe *veículos*, bem como suas operações: dar partida, acelerar, frear e estacionar. Ou seja, o objeto marca: Opel poderá executar as mesmas operações definidas em sua classe, sem que para tanto essas tenham de ser redefinidas para ele.

O conceito de herança é importantíssimo na orientação a objetos pois, como veremos na Seção 4.3.8 — “Aplicação”, a programação fica bastante facilitada. Os códigos escritos na definição da classe, de seus atributos e operações são aproveitados por suas sub-classes e instâncias de objeto, o que reduz o número de linhas de programação, gera maior qualidade e facilita a programação, a verificação de erros e futuras correções.

4.3.6 POLIMORFISMO

Associado ao conceito de herança, o **polimorfismo** permite que um objeto assuma um comportamento diferente daquele definido em sua classe. Polimorfismo significa a capacidade de assumir muitas formas. Isso é importante para permitir o tipo de classificação como a exemplificada anteriormente. Foi visto que uma classe *veículos* possuía as subclasses *utilitários*, *esporte*, *passeio* e *passageiros* e a operação *dar partida*. Tomando-se a operação *dar partida* como exemplo, pode-se dizer que, para dar partida em um veículo de passeio e em um veículo utilitário, pode ser preciso executar tarefas diferentes. Muitas vezes, dependendo do tipo de motor que equipa o veículo e do combustível utilizado, o processo muda significativamente. Considerando-se, assim, que a operação *dar partida* precisa ser adaptada para a classe *utilitários*, reescrevem-se as tarefas para essa operação, que passará a responder de forma diferente apenas para essa classe e para os objetos instanciados por ela.

Em resumo, as classes *utilitários*, *esporte*, *passeio* e *passageiros* herdam todos os atributos e as operações da classe *veículos*. Porém, para a classe *utilitários*, a operação *dar partida* foi modificada para atender a necessidades específicas. Logo, diz-se que a operação *dar partida* apresenta polimorfismo.

4.3.7 ENCAPSULAMENTO

Pensando no modelo de representação de um objeto, pode-se visualizar facilmente a questão do **encapsulamento**. O encapsulamento, também chamado de

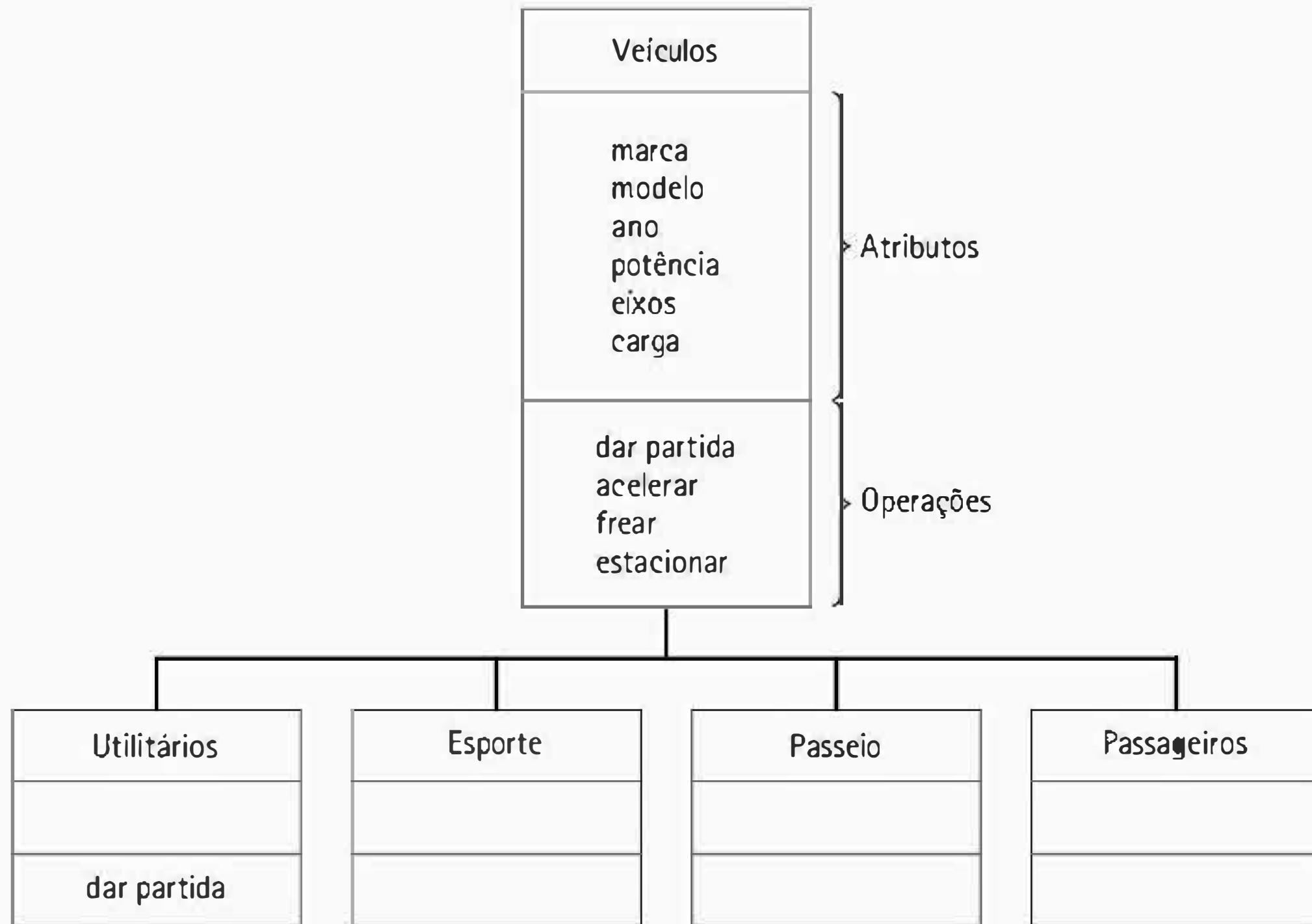


FIGURA 5 | Exemplo de polimorfismo

ocultamento de informações, consiste na separação entre os aspectos externos de um objeto, acessíveis por outros objetos, e os detalhes internos da sua implementação, que ficam ocultos dos demais objetos. O encapsulamento impede que um programa se torne tão interdependente que uma pequena modificação possa provocar grandes efeitos que se propaguem por todo o sistema. É um pouco do conceito de modularização discutido em programação estruturada, lembra-se? Dividia-se o programa em módulos, tratando cada um deles o mais separadamente possível, de modo que fosse ‘quebrada’ a complexidade do problema. Aqui o pensamento é o mesmo, porém a forma de implementação é bastante diferente.

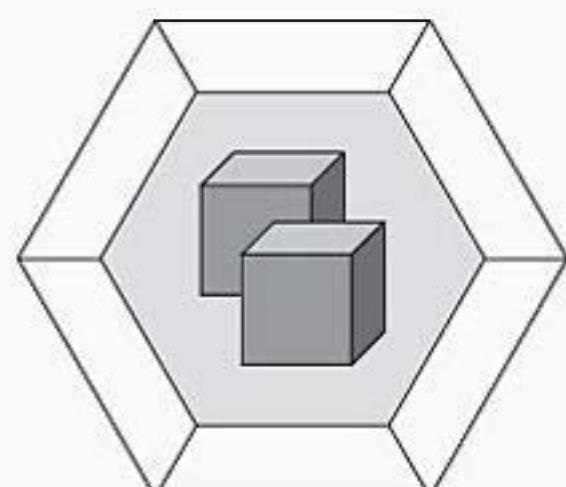


FIGURA 6 | Visualização de um objeto

4.3.8 APLICAÇÃO

Será apresentada nesta seção uma aplicação escrita em Java em que os principais conceitos discutidos em orientação a objetos aparecem. Trata-se de um programa bastante simples que representa um robô que possui os seguintes atributos: nome, número de série, data de fabricação e nível da bateria. Possui, também, as seguintes operações: mudaNome, mostraNome e mostraTudo, que alteram o conteúdo de seus atributos e o exibem. O robô executa algumas operações matemáticas básicas; cada execução consome um pouco da carga de sua bateria.

```

1. public class Robo_superclasse{
2.     String nome;
3.     int serie;
4.     String data;
5.     int bat;
6.     public Robo_superclasse(){
7.     }
8.     public void divide(){
9.     }
10.    public void soma(){
11.    }
12.    public void subtrai(){
13.    }
14.    public void multiplica(){
15.    }
16. }
```

A classe Robo_superclasse será utilizada para demonstração do recurso de herança. A classe Robo, apresentada a seguir, herda os atributos (variáveis) e métodos da classe Robo_superclasse:

```
class Robo extends Robo_superclasse
```

Este recurso permite às classes herdeiras (subclasses) implementar os métodos herdados de diferentes maneiras; a esse recurso chamamos **polimorfismo**.

Neste exemplo não estaremos representando esse recurso, mas ele poderia ser implementado utilizando-se os métodos soma(), subtracao(), multiplicacao() ou divisao() de maneiras diferentes em outras classes herdeiras, por exemplo:

```

class Robo_2 extends Robo_superclasse {
    ...
    public void soma(double a, double b)
    { double c;
        c = (a + b + 5);
    } ...
```

```
1. class Robo extends Robo_superclasse
2. {
3.     public Robo(){
4.         nome = "XPT11";
5.         serie = 0000;
6.         data = "00/00/0000";
7.         bat = 5;
8.     }
9.     public Robo(String n, int s, String d){
10.        nome = n;
11.        serie = s;
12.        data = d;
13.        bat = 5;
14.    }
15.    public void mudaNome(String novo_nome){
16.        this.nome = novo_nome;
17.    }
18.    public String mostraNome(){
19.        return nome;
20.    }
21.    public void mostraTudo(){
22.        System.out.println("Robo chama-se: " + nome);
23.        System.out.println("Serie: " + serie);
24.        System.out.println("Data: " + data);
25.        System.out.println("Bateria: " + bat);
26.    }
27.    public void soma(int a, int b){
28.        System.out.println("O nivel da bateria esta' em: " + bat);
29.        if (bat >=1){
30.            int c;
31.            c = a + b;
32.            System.out.println("O resultado da operacao e': " + c);
33.            bat--;
34.            System.out.println("Nivel da bateria: " + bat);
35.        }
36.        if (bat == 1){
37.            System.out.println("Bateria fraca, recarregar!");
38.        }
39.        if (bat == 0){
40.            System.out.println("Robo danificado!");
41.        }
42.    }
43.    public void subtrai(int a, int b){
44.        if (bat >=1){
45.            int c;
46.            c = a - b;
47.            System.out.println("O resultado da operacao e': " + c);
48.            bat--;
49.        }
50.    }
51. }
```

```
49.         System.out.println("Nivel da bateria: " + bat);
50.     }
51.     if (bat == 1) {
52.         System.out.println("Bateria fraca, recarregar!");
53.     }
54.     if (bat == 0) {
55.         System.out.println("Robo danificado!");
56.     }
57. }
58. public void multiplica(int a, int b){
59.     if (bat >=1){
60.         int c;
61.         c = a * b;
62.         System.out.println("O resultado da operacao e' : " + c);
63.         bat--;
64.         System.out.println("Nivel da bateria: " + bat);
65.     }
66.     if (bat == 1) {
67.         System.out.println("Bateria fraca, recarregar!");
68.     }
69.     if (bat == 0){
70.         System.out.println("Robo danificado!");
71.     }
72. }
73. public void divide(int a, int b){
74.     if (bat >=1){
75.         int c;
76.         c = a / b;
77.         System.out.println("O resultado da operacao e' : " + c);
78.         bat--;
79.         System.out.println("Nivel da bateria: " + bat);
80.     }
81.     if (bat == 1) {
82.         System.out.println("Bateria fraca, recarregar!");
83.     }
84.     if (bat == 0){
85.         System.out.println("Robô danificado!");
86.     }
87. }
88. }
```

Nesse exemplo, pode-se verificar a criação de uma classe denominada Robo, que herda os atributos e métodos da superclasse Robo_superclasse (linha 1) que possui os atributos declarados nas linhas 3 a 6. Os chamados **métodos construtores** são responsáveis pela criação das instâncias de classe, ou seja, novos objetos com as características da classe Robo. Foram definidas duas formas de criação de instâncias: uma sem passagem de parâmetros (linha 3) e outra com a passagem de parâmetros definindo nome, série e data (linha 9).

ATENÇÃO!

A palavra Robo refere-se ao nome da classe e deve ser escrita sem acento.

NOTA:

Método é a implementação ou a codificação em uma linguagem de programação de uma operação de um objeto.

O método mudaNome (linha 15) permite a alteração do nome do robô. Esse método recebe como parâmetro o novo nome que o robô deverá assumir. Esse método demonstra a característica de encapsulamento de dados da metodologia, ou seja, sómente é possível alterar o nome de um robô acessando-se esse método. Outros atributos de Robo, como o número de série, por exemplo, não poderão ser alterados. Uma vez definido quando da instanciação de um novo robô, o número de série não poderá mais ser alterado, visto que não há um método que permita esse tipo de alteração. Preserva-se, assim, a integridade dos dados.

Robo também executa algumas operações aritméticas como, por exemplo, a soma de dois números inteiros fornecidos, exibindo o resultado. Isso foi implementado nas linhas 27 a 42. A exibição do resultado se dá na linha 32, onde se mostra a literal: "O resultado da operação é: ", seguida do valor da variável c.

Quando da execução de uma operação aritmética, a bateria do robô, que se inicia com a carga 5, é decrementada, o que denota um desgaste. Isso é feito pelo comando na linha 38. Após a execução de uma seqüência de quatro operações aritméticas, o Robo atinge a carga da bateria igual a 0 (zero). Nesse caso, é exibida a mensagem: "Robo danificado!", por meio dos comandos nas linhas 39 a 41.

A seqüência de código a seguir executa uma espécie de teste da classe Robo, no qual são criadas duas instâncias da classe, R1 e R2, são chamados alguns métodos e são executadas algumas operações aritméticas.

```

1. class TestaRobo
2. {
3.     public static void main(String[] args){
4.         System.out.println("Testando Robo");
5.         Robo R1 = new Robo();
6.         System.out.println("Robo: " + R1.mostraNome());
7.         R1.soma(3, 2);
8.         R1.subtrai(8, 2);
9.         R1.multiplica(3, 4);
10.        R1.divide(8, 2);
11.        R1.divide(6, 3);
12.
13.        Robo R2 = new Robo("D288", 2, "05/11/2002");
14.        System.out.println("Testando Robo");
15.        System.out.println("Robo: " + R2.mostraNome());
16.        R2.soma(5, 4);
17.        R2.multiplica(3, 5);
18.        R2.mostraTudo();

```

```
19.    }
20. }
```

A instanciação de um objeto da classe Robo é feita por meio do código que está nas linhas 5 e 13; em cada caso, foi usado um dos dois métodos construtores possíveis. Deve-se notar a chamada de um método, como o que exibe o nome do robô (linha 6), utilizando-se a notação: Objeto.método, no caso, R1.mostraNome().

4.4 EXERCÍCIOS PARA FIXAÇÃO

1. Faça um comparativo entre a programação estruturada e a programação orientada a objetos e cite as vantagens de cada uma.
2. Quais são as principais características da programação estruturada?
3. Quais são as principais características da programação orientada a objetos?
4. Quais são as principais características da programação linear?
5. Cite alguns exemplos de linguagens de programação orientada a objetos, estruturada e linear.

4.5 EXERCÍCIOS COMPLEMENTARES

1. Descreva alguns exemplos de atividades nas quais seja possível o uso do recurso de polimorfismo.
2. Descreva alguns exemplos de atividades nas quais seja possível o uso do recurso de encapsulamento.
3. Descreva alguns exemplos de atividades nas quais seja possível o uso do recurso de herança.

CONSTRUÇÃO DE ALGORITMOS: ESTRUTURAS DE CONTROLE

- ▶ Entrada e saída de dados
- ▶ Estruturas de seleção (de decisão ou de desvio)
- ▶ Estruturas de repetição
- ▶ Exemplos e exercícios para fixação



OBJETIVOS:

Abordar as técnicas para entrada e saída de dados e as estruturas para controle do fluxo de dados em pseudocódigo, fluxograma e na linguagem de programação Java.

Imagine que pretendemos construir um aparelho para realizar uma ação qualquer na casa do comprador. Como não moramos na casa dos possíveis compradores, não sabemos se o aparelho será utilizado da maneira que planejamos e, portanto, devemos prever os possíveis comportamentos do aparelho para que cumpra suas tarefas sem problemas para o comprador. Quando fazemos um programa, devemos saber construí-lo para prever qualquer intenção que o usuário tenha ao utilizar esse programa.

Entrada e saída de dados



É importante ressaltar a seqüência de fatos que fundamentam a lógica computacional: a **entrada** dos dados que serão **processados** para obter a **saída**. Os dados que entram em processamento sofrem transformações resultantes do processo e uma saída é produzida, representando a solução de um problema. Vamos fazer uma breve análise desses dois aspectos de um programa.

5.1 ENTRADA

A entrada elementar de dados é feita por meio do teclado (dispositivo-padrão) e é representada por:

Ler (variável)

Para uma variável inteira, por exemplo, esse comando procura uma seqüência de caracteres que representem os dígitos de um inteiro eventualmente precedidos por um sinal + ou -. Nesse caso, são descartados eventuais caracteres brancos e, a partir do primeiro caractere não branco, a rotina de leitura assume que encontrou uma cadeia de caracteres que está de acordo com a sintaxe dos inteiros.

Se isso não acontece, ocorre um *erro fatal* e a execução do programa é interrompida. Se, por outro lado, a rotina encontra um caractere que atenda à sintaxe de um inteiro, ela continua a consumir caracteres até que encontre algo diferente, como um caractere branco, por exemplo.

Durante o processo, a seqüência de caracteres que satisfaz a sintaxe de um inteiro é convertida em um valor binário. Ao final do processo, o valor binário resultante é armazenado na posição correspondente à variável inteira para a qual a rotina de entrada procurou um valor.

Quando falamos que caracteres brancos são descartados, estamos apenas exemplificando. Isso ocorre também com caracteres de formatação de texto como o de tabulação, o de mudança de linha e o de ‘retorno de carro’.

Ao digitarmos dados pelo teclado, eles são ‘echoados’ na tela do monitor do computador, isto é, são mostrados na tela conforme vão sendo digitados. Enquanto não se pressiona uma tecla de mudança de campo (ENTER para DOS ou TAB para telas gráficas), o processo de leitura não é disparado e o programa suspende a execução do comando de leitura que está solicitando dados do usuário. Ao ocorrer o disparo pelo pressionar da tecla ENTER (ou TAB), a execução do programa é retomada nesse ponto.

No caso particular da linguagem Java, todas as variáveis lidas por meio do teclado são reconhecidas como caracteres da tabela UNICODE. Como a linguagem é rigorosa quanto aos dados que irá manipular, para ler uma variável de um tipo qualquer, deve-se utilizar um processo denominado coerção, que nada mais é do que a produção de um valor com o tipo diferente do original. Essa coerção é realizada por meio de recursos especiais do Java, que podem ser consultados no Anexo.

5.2 SAÍDA

É claro que um computador e toda a sua programação não seriam de nenhuma utilidade se o programa não mostrasse o resultado das operações. O dispositivo-padrão de saída é o monitor do computador, e essa saída é representada com o comando:

Mostrar (variável)

A maioria das linguagens de programação possui recursos que permitem fazer uma formatação básica da saída de tela com comandos que escrevem na mesma linha ou ‘pulam’ para a linha inferior.

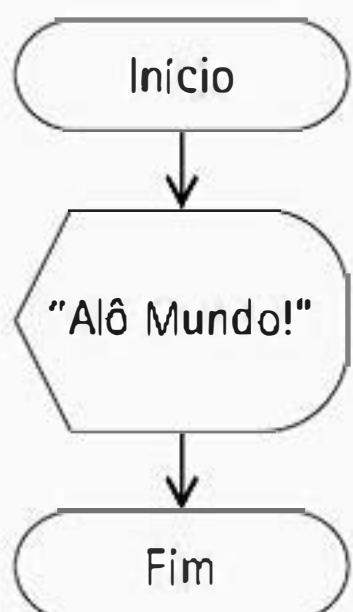
EXEMPLO 5.1: Como é costume, o primeiro programa que se aprende a fazer em qualquer linguagem é um aplicativo que mostra uma frase na tela: “Alô Mundo!” Nesse programa não é necessário o uso de variáveis.

Pseudocódigo:

1. Algoritmo Primeiro
2. Início
3. Mostrar (“Alô Mundo!”)
4. Fim.

Na linha 1 o algoritmo é identificado. As linhas 2 e 4 representam o início e o fim do algoritmo, e a linha 3 é a ação que o algoritmo tem que realizar.

Fluxograma:



Java:

```

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

Sobre o código:

1. `class Primeiro` indica o nome do programa. No caso do Java, todos os programas são classes.

2. `public static void main (String args [])` indica o bloco de instruções que serão executadas seqüencialmente quando o programa for requisitado pelo usuário. Todo aplicativo escrito em Java deve possuir um bloco indicado dessa maneira para poder ser executado.
3. `System.out.println("Alô Mundo!")` é a instrução que ordena a exibição da frase na saída do sistema.

As chaves { e } indicam início e fim de bloco, respectivamente.

CUIDADO!

A linguagem Java é sensível a maiúsculas e minúsculas. Isso quer dizer que as palavras reservadas devem ser escritas exatamente como são definidas! `class` ≠ `Class`, `public` ≠ `Public` ≠ `PUBLIC` e assim por diante.

EXEMPLO 5.2:

O programa a seguir pergunta qual é o nome do usuário e o escreve novamente na tela.

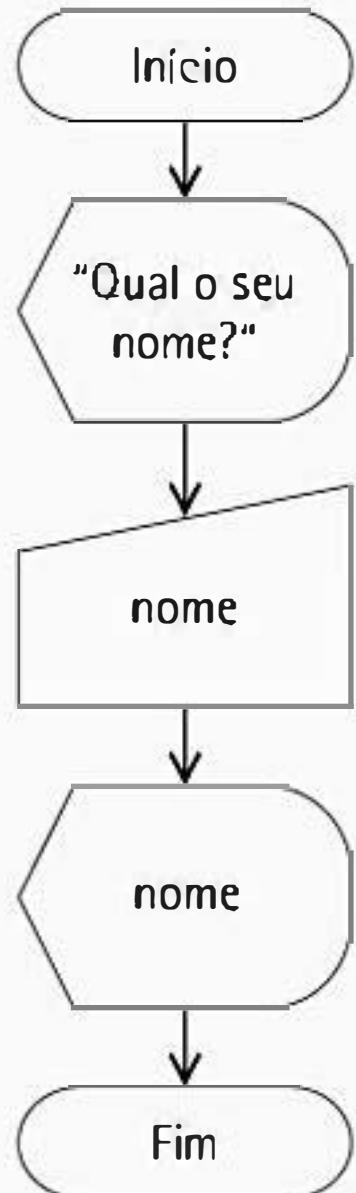
Pseudocódigo:

1. Algoritmo ExEntrada
2. Var
3. nome : literal
4. Início
5. Mostrar ("Qual o seu nome?")
6. Ler (nome)
7. Mostrar (nome)
8. Fim.

Neste exercício é necessário o uso de uma variável para representar o armazenamento do nome do usuário; a declaração dessa variável está sendo feita na linha 3. As linhas 5, 6 e 7 são as ações necessárias para a realização da tarefa; na linha 5 está sendo exibida a mensagem “Qual o seu nome?”; as mensagens sempre devem estar entre aspas.

Na linha 6 é indicado que deve ser lido (fornecido) um valor para a variável nome. Na linha 7 o conteúdo da variável nome está sendo exibido.

Fluxograma:



Java:

```

1. import java.io.*;
2. class ExEntrada {
3.     public static void main (String args []){
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
6.             (System.in));
7.         String nome;
8.         try {
9.             System.out.println("Qual o seu nome?");
10.            nome = entrada.readLine();
11.            System.out.println(nome);
12.        } catch (Exception e) {
13.            System.out.println("Ocorreu um erro durante a
14. leitura!");
15.        }
16.    }
17. }
  
```

Sobre o código:

1. A cláusula `import` indica a utilização do pacote de entrada e saída de dados (`java.io.*`) da linguagem Java nesse programa. Embora isso seja desnecessário para a saída de dados, a linguagem requer um tratamento de entrada de dados para sua leitura correta.
2. e 3. `class ExEntrada` indica o nome do programa e, novamente, `public static void main (String args [])` indica o bloco de instruções

que serão seqüencialmente executadas quando o programa for requisitado pelo usuário.

4. e 5. Não se assuste! Nessas linhas estão a declaração e a criação da variável entrada que será utilizada para a leitura do teclado. Isso é necessário para garantir que a leitura seja armazenada na memória com o formato de uma seqüência de caracteres (*String*) para que seu valor possa ser manipulado sempre da mesma forma. A vantagem dessa sintaxe é que o programador pode utilizar apenas uma variável de entrada e depois convertê-la para os valores desejados quantas vezes forem necessárias.
6. A variável nome está sendo declarada para receber e armazenar na memória o valor escrito pelo usuário. Como se deseja a entrada de um texto, ela é declarada como do tipo *String*.
- 7., 11., 12. e 13. Definem um bloco de tratamento de erros característico da linguagem Java. Sempre que houver entrada de dados na linguagem Java, um tratamento de erros similar será necessário. Para os fins da lógica deste livro, o tratamento será utilizado sempre dessa forma.
É recomendável que o programa exiba uma frase solicitando ao usuário as informações necessárias para a sua execução.
9. A variável nome recebe, por meio do método *readLine()*, o valor que o usuário escreveu. A variável entrada é um objeto que permite a utilização desse método para a leitura. Essa maneira de chamar métodos é típica da orientação a objetos.
10. Após a leitura da variável, ela é mostrada na tela como saída de dados.
14. e 15. As chaves dessas linhas fecham os blocos do método e da classe, respectivamente.

EXEMPLO 5.3: O programa a seguir realiza a soma de dois números inteiros dados pelo usuário.

Pseudocódigo:

1. Algoritmo ExSoma
2. Var
3. valor1, valor2, soma : inteiro
4. Início
5. Mostrar ("Qual o primeiro valor?")
6. Ler (valor1)
7. Mostrar ("Qual o segundo valor?")
8. Ler (valor2)
9. soma ← valor1 + valor2
10. Mostrar (soma)
11. Fim.

Fluxograma:



Neste exemplo, além da exibição de mensagens e entrada de dados, ocorre também um processamento determinado pela operação aritmética de adição entre as variáveis `valor1` e `valor2` e pela atribuição do resultado à variável `soma` (linha 9).

Java:

```

1. import java.io.*;
2. class ExSoma {
3.     public static void main (String args []) {
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
6.             (System.in));
7.         int valor1, valor2, soma;
8.         try {
  
```

```
8.         System.out.println("Qual o primeiro valor?");
9.         valor1 = Integer.parseInt(entrada.readLine());
10.        System.out.println("Qual o segundo valor?");
11.        valor2 = Integer.parseInt(entrada.readLine());
12.        soma = valor1 + valor2;
13.        System.out.println(soma);
14.    } catch (Exception e) {
15.        System.out.println("Ocorreu um erro durante a
16.        leitura!");
17.    }
18. }
```

Repare que as linhas em itálico são exatamente iguais às do programa anterior. Essas linhas são características da linguagem e não afetam a lógica do problema.

Sobre o código:

Na linha 6 são declaradas três variáveis do tipo inteiro para armazenar os valores inteiros dados pelo usuário e realizar a soma.

As linhas 9 e 11 recebem os valores lidos do console na forma de texto e os convertem para números inteiros utilizando o método `Integer.parseInt()`, também específico da linguagem Java. Note que, no caso de o usuário digitar um número real ou uma letra, o programa não efetuará um erro fatal devido ao tratamento de erro, mas exibirá uma mensagem de erro para o usuário, sem realizar a conta. Essa mensagem pode ser personalizada para cada programa.

Finalmente, as linhas 12 e 13 realizam a soma desejada e mostram o resultado na tela.

5.3 ESTRUTURAS DE SELEÇÃO OU DECISÃO

As estruturas de seleção ou decisão são utilizadas quando existe a necessidade de verificar condições para a realização de uma instrução ou de uma seqüência de instruções. Os testes de seleção também podem ser utilizados para verificar opções de escolha. A seguir são apresentados exemplos para os dois casos.

Suponha que uma pessoa esteja jogando um jogo de computador:

1. Para que o jogador passe de uma fase (etapa) para a fase seguinte, é necessário que se verifique se ele atingiu a pontuação exigida. Assim, existe uma condição para a realização de uma seqüência de instruções para liberar o acesso à próxima fase do jogo.
2. Ao final do jogo, uma pergunta é feita: “Deseja continuar jogando?” O jogador poderá escolher entre as respostas **sim** ou **não**.

As estruturas de seleção podem ser do tipo simples, composto ou encadeado.

5.4 ESTRUTURAS DE SELEÇÃO SIMPLES

São utilizadas para verificar se dada condição é atendida: se for, um conjunto de instruções deverá ser executado; se não for, o fluxo da execução do algoritmo seguirá após o fim do bloco de decisão.

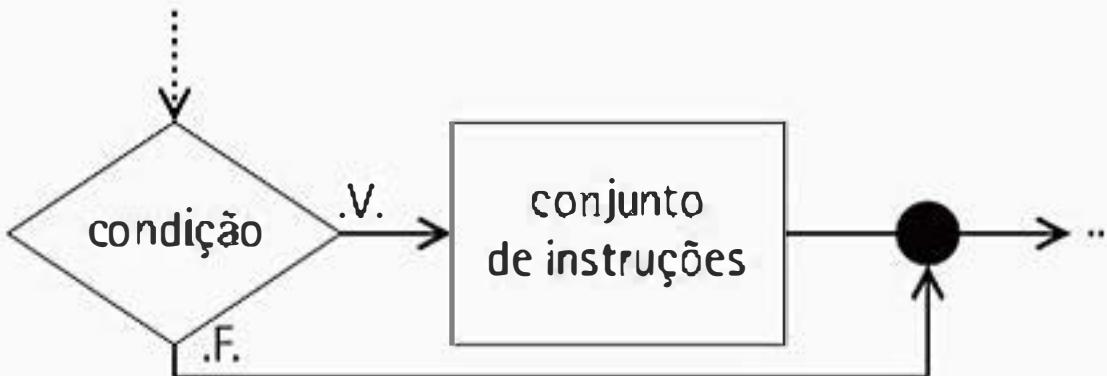
NOTA:

Toda condição pode ser encarada como uma pergunta que pode ter a resposta verdadeiro (.v.) ou falso (.f.).

Pseudocódigo:

```
Se (condição) então [início do bloco de decisão]
    conjunto de instruções
Fim-Se [fim do bloco de decisão]
```

Fluxograma:



Java:

```
if (condição) {
    <conjunto de instruções>;
}
```

NOTA:

Quando o teste de condição resultar verdadeiro, sempre será executado o primeiro conjunto de instruções encontrado. Caso contrário, isto é, se a condição resultar falso, será realizado o segundo conjunto de instruções, ou seja, o conjunto de instruções após o senão. (Vide o exemplo no item “Estruturas de seleção compostas”.)

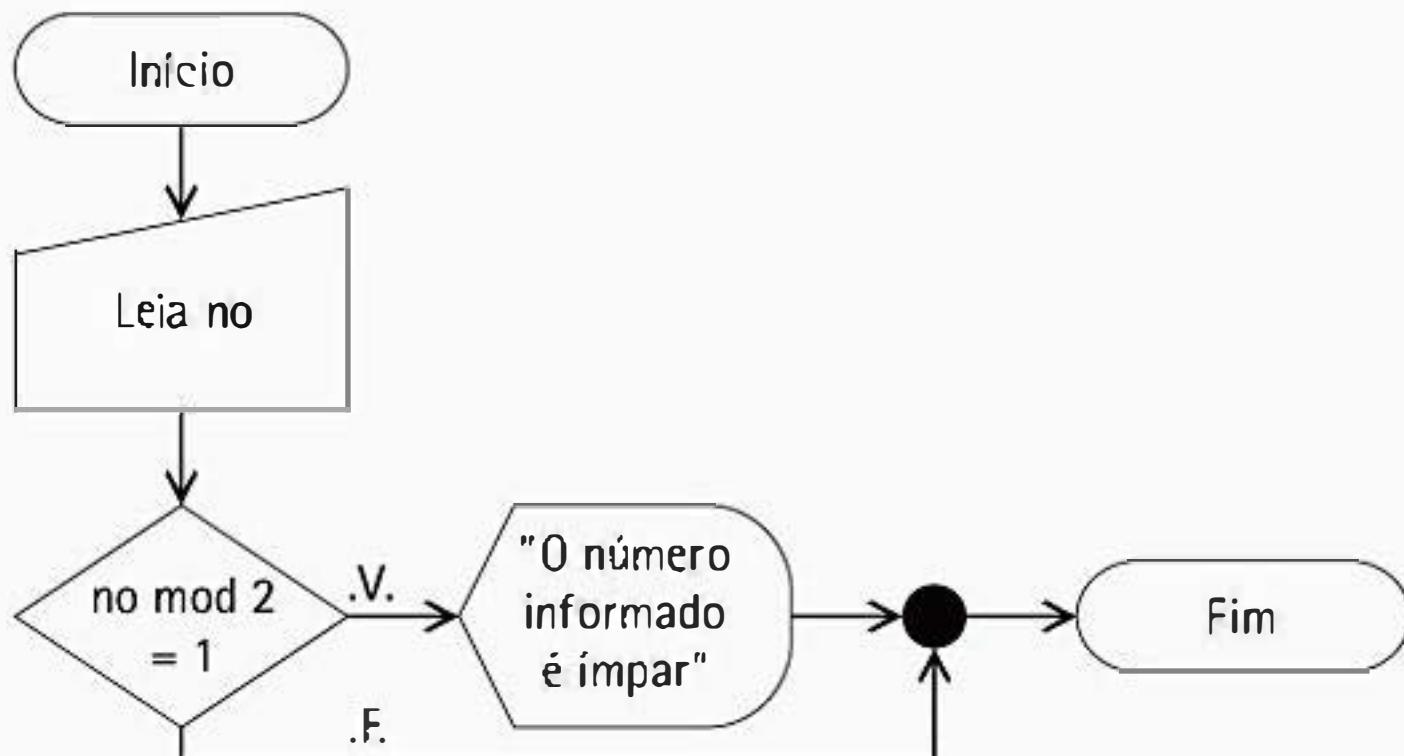
EXEMPLO 5.4: Verificar se um número fornecido pelo usuário é ímpar. Se for, exibir a mensagem: “● número informado é ímpar”.

Pseudocódigo:

1. Algoritmo no_impar
2. Var
3. no: inteiro
4. Início
5. Ler (no)
6. Se (no mod 2 = 1) Então
7. Mostrar ("O número informado é ímpar")
8. Fim-Se
9. Fim.

Neste exemplo, na linha 6 é feita a avaliação da condição. Como só existe uma instrução a ser realizada (mostrar a mensagem “O número informado é ímpar”), então esta é uma estrutura de seleção simples.

Fluxograma:



Java:

```

1. import java.io.*;
2. class NumImpar {
3.     public static void main (String args []){
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
(System.in));
6.         int numero ;
7.         try {
8.             System.out.println("Qual o número? ");
9.             numero = Integer.parseInt(entrada.readLine());
10.            if (numero % 2 == 1) {
11.                System.out.println ("O número informado é ímpar");
12.            }
13.        } catch (Exception e) {
14.            System.out.println("Ocorreu um erro durante a
leitura! ");
15.        }
16.    }
17. }
  
```

Sobre o código:

Na linha 6 é declarada a variável de tipo inteiro que receberá, na linha 9, o valor digitado pelo usuário e que será avaliado no programa.

Na linha 10, a condicional avalia se o número é ímpar ao verificar se o resto da divisão desse número por 2 é igual a 1. Se isso for verdade, então a frase escrita na linha 11 será mostrada na tela.

Observe que `%` é o operador que determina o resto da divisão e `==` é o operador que compara o resultado dessa operação com 1. Em Java, o operador `=` determina atribuição de valor e o operador `==` determina comparação de valores.

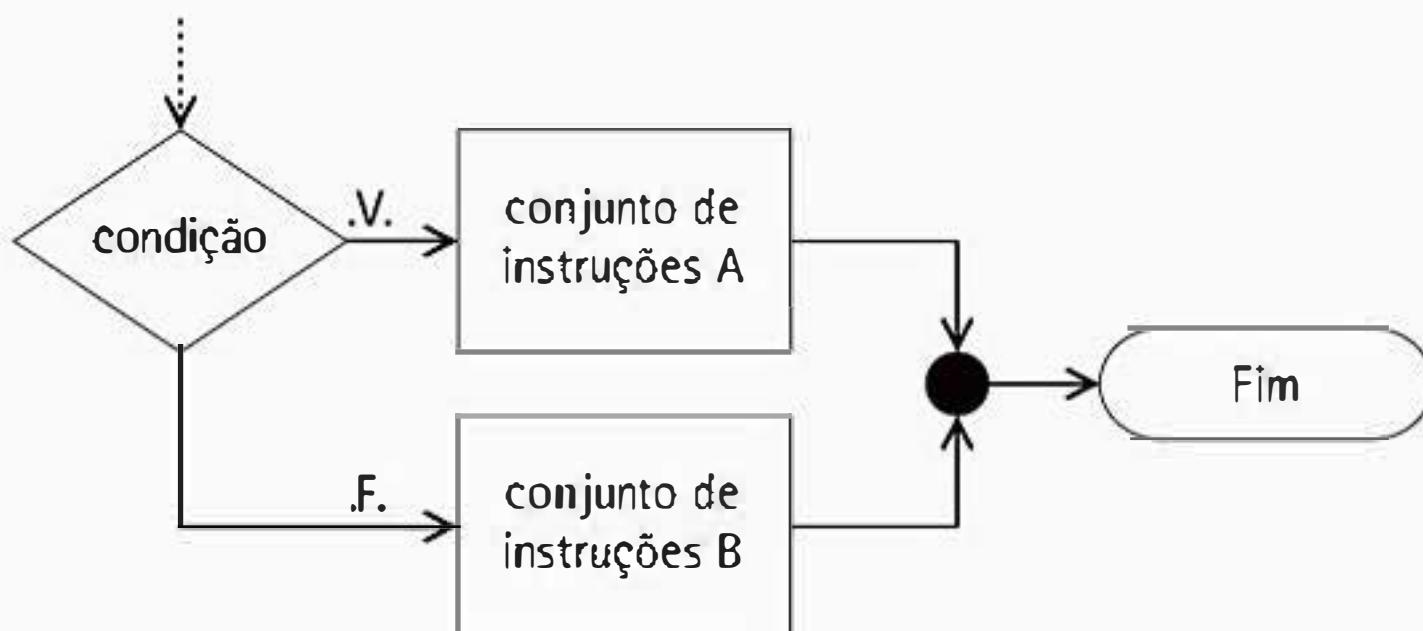
5.5 ESTRUTURAS DE SELEÇÃO COMPOSTAS

A **estrutura de seleção composta** prevê uma condição com dois conjuntos de instruções para serem realizados de acordo com a avaliação da resposta: um bloco de instruções para resposta verdadeiro e um bloco de instruções para resposta falso.

Pseudocódigo:

Se (condição) então	
conjunto de instruções A	[conjunto de instruções que será realizado se o teste de condição resultar verdadeiro]
Senão	
conjunto de instruções B	[conjunto de instruções que será realizado se o teste de condição resultar falso]
Fim-Se	

Fluxograma:



Java:

```

if (condição) {
    <conjunto de instruções A>;
} else {
    <conjunto de instruções B>;
}
  
```

EXEMPLO 5.5: A empresa XKW Ltda. concedeu um bônus de 20 por cento do valor do salário a todos os funcionários com tempo de trabalho na empresa igual ou superior a cinco anos e de dez por cento aos demais. Calcular e exibir o valor do bônus.

Para resolver o problema, é necessário conhecer o valor do salário e o tempo de serviço do funcionário. Para isso, serão utilizadas as variáveis `salario` e `tempo`, que representarão esses valores. Para armazenar o valor do bônus, será utilizada a variável `bonus`.

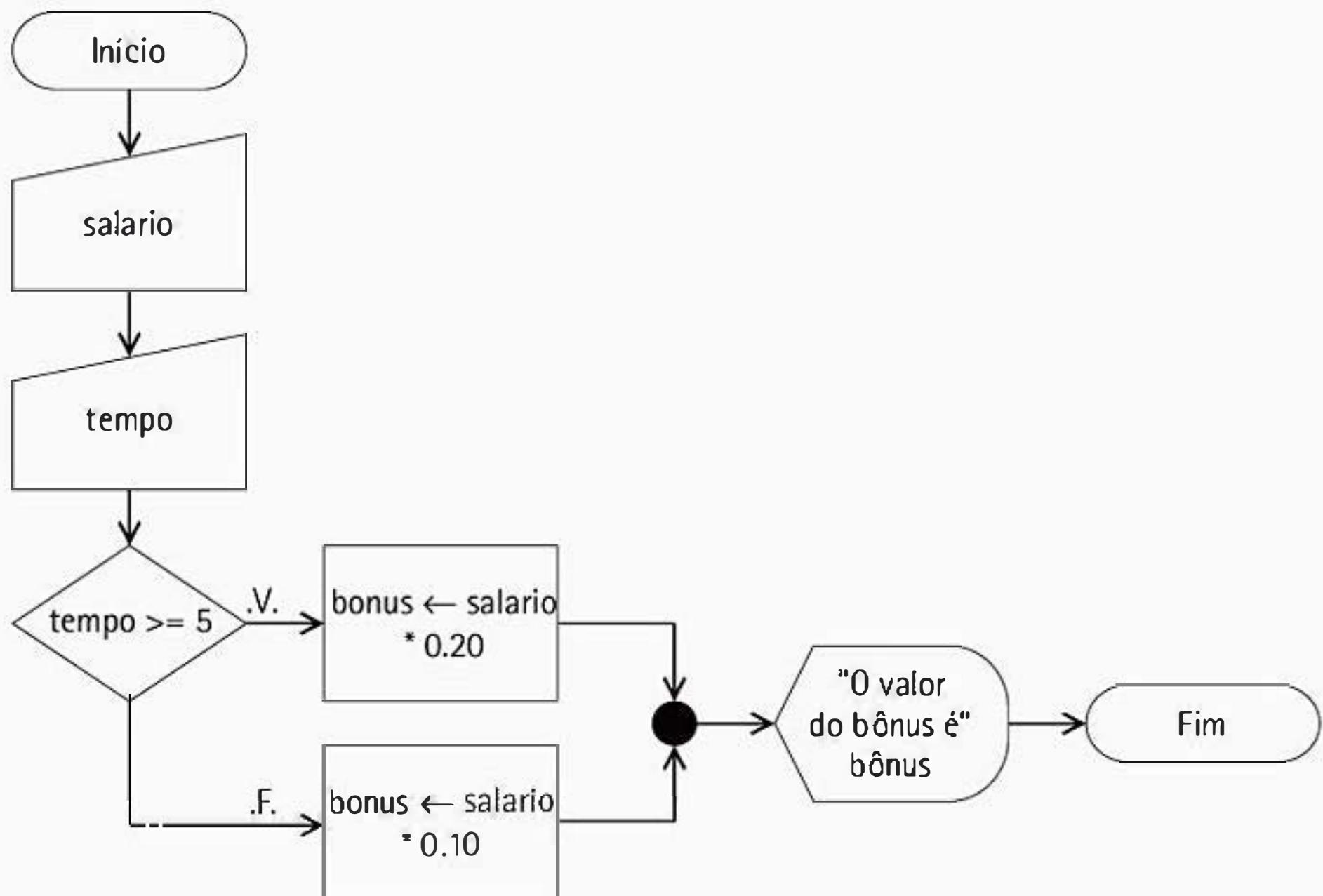
Pseudocódigo:

```

1. Algoritmo Premio
2. Var
3.     salario, bonus: real
4.     tempo: inteiro
5. Início
6.     Ler (salario)
7.     Ler (tempo)
8.     Se (tempo >= 5) então
9.         bonus ← salario * 0.20
10.    Senão
11.        bonus ← salario * 0.10
12.    Fim-Se
13.    Mostrar ("O valor do bônus é", bonus)
14. Fim.

```

Na linha 8 está sendo feito o teste para verificação da condição que foi estabelecida no enunciado; esta é uma condição com resposta verdadeira e uma instrução para resposta falsa, por isso é uma condição composta.

Fluxograma:**Java:**

```

1. import java.io.*;
2. class Premio {
3.     public static void main (String args []) {
4.         BufferedReader entrada;

```

```

5.         entrada = new BufferedReader(new InputStreamReader
6.             (System.in));
7.         float salario, bonus;
8.         int tempo;
9.         try {
10.             System.out.println("Qual o salário?");
11.             salario = Float.parseFloat(entrada.readLine());
12.             System.out.println("Quanto tempo está na empresa?");
13.             tempo = Integer.parseInt(entrada.readLine());
14.             if (tempo >= 5) {
15.                 bonus = salario * 0.20f;
16.             } else {
17.                 bonus = salario * 0.10f;
18.             }
19.             System.out.println ("O valor do bônus é: " + bonus);
20.         } catch (Exception e) {
21.             System.out.println("Ocorreu um erro durante a
22.                 leitura!");
23.         }

```

Sobre o código:

Nas linhas 6 e 7 são declaradas as variáveis necessárias para o programa. Observe que o salário e o bônus são variáveis do tipo `real` (no Java, `float`) e o tempo é uma variável do tipo `inteiro`. Nas linhas 10 e 12, onde se fará a leitura desses valores, isso é observado pela conversão adequada dos valores de entrada com as funções `Float.parseFloat()` e `Integer.parseInt()`.

Nas linhas 13 a 17 é realizada a condicional composta que fornece o resultado desejado, como no algoritmo em português estruturado. Observe a letra `f` no final dos valores multiplicados: essa é outra característica da linguagem Java que garante que o tipo resultante será um valor `real` do tipo `float`.

5.6 ESTRUTURAS DE SELEÇÃO ENCADEADAS

Uma **estrutura de seleção encadeada** é uma seqüência de testes de seleção, os quais serão executados ou não de acordo com o resultado das condições e de acordo com o encadeamento dos testes, isto é, um teste de seleção pode ter dois conjuntos de instruções, conforme visto na Seção “Estruturas de seleção compostas”, um para resultado `verdadeiro` e outro para `falso`; porém, esses conjuntos de instruções podem conter outros testes de seleção, que por sua vez também podem conter outros e assim por diante.

Pseudocódigo:

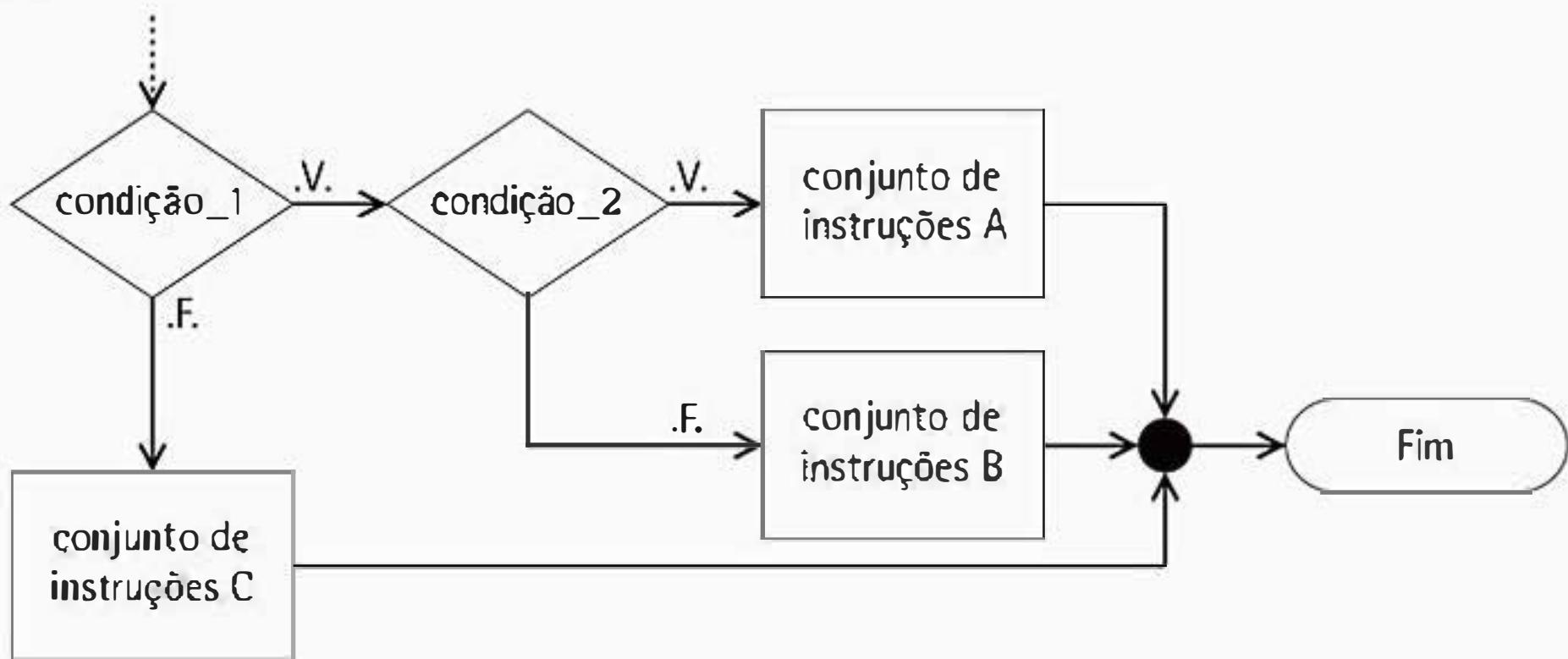
```

Se (condição_1) então
    Se (condição_2) então
        conjunto de instruções A
    Senão
        conjunto de instruções B
    Fim-Se
Senão
    conjunto de instruções C
Fim-Se

```

NOTA:

No modelo acima, se a condição_1 resultar verdadeiro, então será realizado o teste da condição_2; se esse teste resultar verdadeiro, será realizado o conjunto de instruções A; se resultar falso, será realizado o conjunto de instruções B. Se o teste da condição_1 resultar falso, será realizado o conjunto de instruções C.

Fluxograma:**Java:**

```

if (<condição_1>) {
    if (<condição_2>) {
        <conjunto de instruções A>
    } else {
        <conjunto de instruções B>
    }
} else {
    <conjunto de instruções C>
}

```

EXEMPLO 5.6: Faça um algoritmo que receba três valores que representarão os lados de um triângulo e serão fornecidos pelo usuário. Verifique se os valores formam um triângulo e classifique esse triângulo como:

eqüilátero – três lados iguais;
isósceles – dois lados iguais;
escaleno – três lados diferentes.

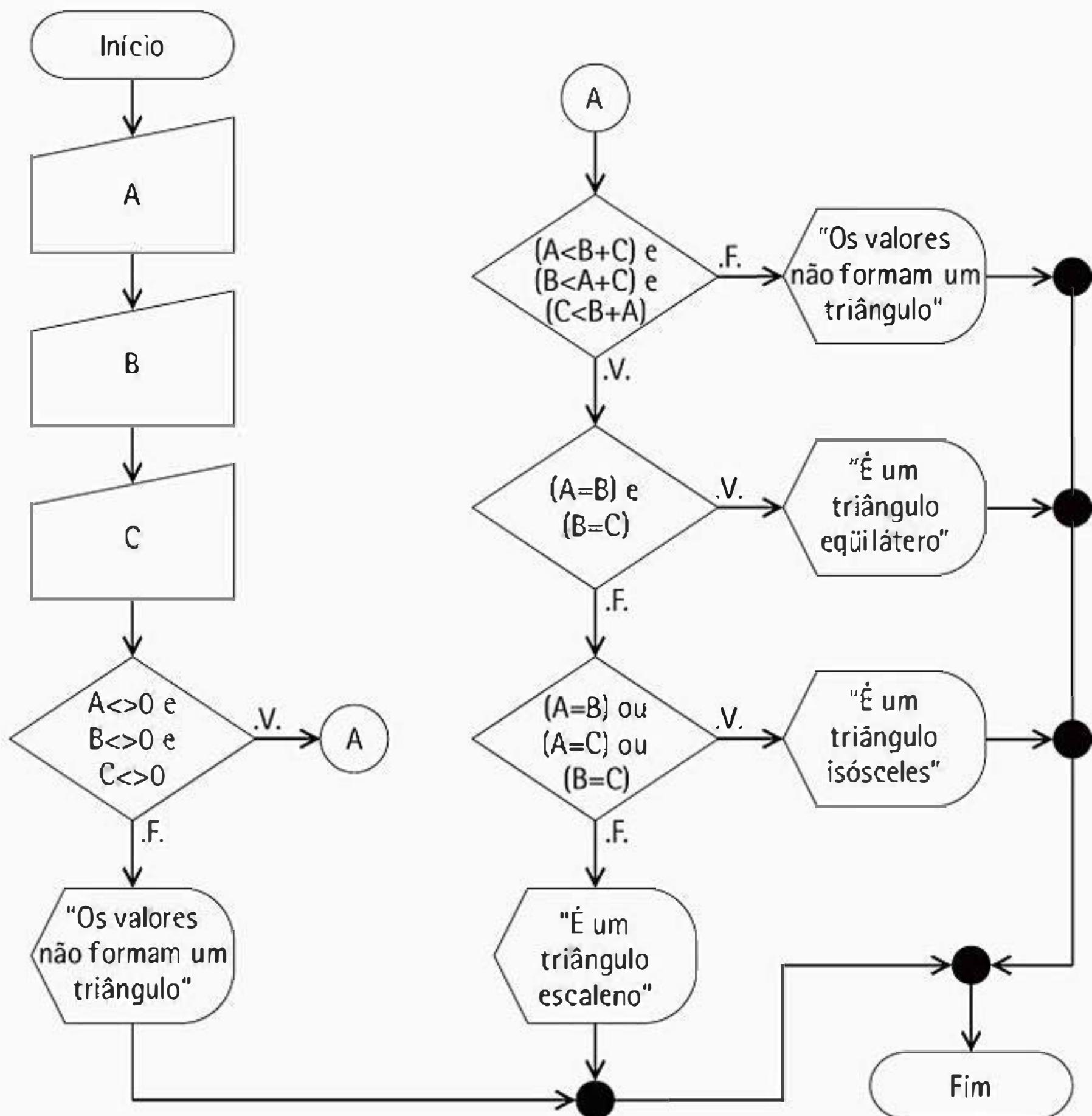
Lembre-se de que, para formar um triângulo:
nenhum dos lados pode ser igual a zero;
um lado não pode ser maior do que a soma dos outros dois.

Pseudocódigo:

```

1. Algoritmo triangulo
2. Var
3.     A,B,C: inteiro
4. Inicio
5.     Ler (A, B, C)
6.     Se (A <> 0) .e. (B <> 0) .e. (C <> 0) então
7.         Se (A<B+C) .e. (B<A+C) .e. (C<A+B) então
8.             Se (A = B) .e. (B = C) então
9.                 Mostrar ("É um triângulo eqüilátero")
10.            Senão
11.                Se (A = B) .ou. (A = C) .ou. (C = B) então
12.                    Mostrar ("É um triângulo isósceles")
13.                Senão
14.                    Mostrar ("É um triângulo escaleno")
15.                Fim-Se
16.            Fim-Se
17.        Senão
18.            Mostrar ("Os valores não formam um triângulo")
19.        Fim-Se
20.    Senão
21.        Mostrar ("Os valores não formam um triângulo")
22.    Fim-Se
23. Fim.

```

Fluxograma:**Java:**

```

1. import java.io.*;
2. class triangulo{
3.     public static void main (String args[]){
4.         int A, B, C;
5.         BufferedReader entrada;
6.         entrada = new BufferedReader(new
InputStreamReader(System.in));
7.         try{
8.             System.out.println ("Lado A =");
9.             A = Integer.parseInt (entrada.readLine ());
10.            System.out.println ("Lado B =");
11.            B = Integer.parseInt (entrada.readLine ());
12.            System.out.println ("Lado C =");
13.            C = Integer.parseInt (entrada.readLine ());
14.            if (A != 0 && B != 0 && C != 0){
15.                if (A + B > C && A + C > B && B + C > A){
16.                    if (A != B && A != C && B != C){
  
```

```

17.                     System.out.println("Escaleno");
18.     }else{
19.         if (A == B && B == C){
20.             System.out.println("Equilatero");
21.         }else{
22.             System.out.println("Isosceles");
23.         }
24.     }
25. }else{
26.     System.out.println("Nao forma um triangulo");
27. }
28. }else{
29.     System.out.println("Nao forma um triangulo");
30. }
31. } catch(Exception e) {
32.     System.out.println("Erro de leitura");
33. }
34. }
35. }
```

Na resolução do exemplo 5.6 são feitos testes de condição encadeados, isto é, testes de condição do conjunto de instruções para uma resposta (ou para ambas) que contém um outro teste de condição.

Na linha 6 do pseudocódigo e na linha 14 do programa ocorre um teste de condição para verificar se os valores fornecidos podem formar um triângulo. Assim, se essa primeira condição for atendida, isto é, se a resposta for verdadeira, um outro teste será realizado nas linhas 7 do pseudocódigo e 15 do programa; então, novamente, se essa condição for verdadeira, serão realizados os testes das linhas 8 e 11 do algoritmo e 16 e 19 do programa para classificar o triângulo.

5.7 ESTRUTURAS DE SELEÇÃO DE MÚLTIPLA ESCOLHA

Uma estrutura de seleção de múltipla escolha é uma estrutura de seleção que funciona como um conjunto de opções para escolha. É também denominada **estrutura de seleção homogênea**. Existem duas maneiras para representá-la: utilizando o encaadeamento da instrução `se` e utilizando a instrução `escolha caso`. A segunda opção é a mais indicada.

Estrutura com condicionais encadeadas

```

Se variável = Tal_Coisa_1
    Faça instrução a
    Senão
        Se variável = Tal_Coisa_2
            Então instrução b
```

```

    Senão
        Se variável = Tal_Coisa_3
            Então instrução c
        Senão instrução d
    Fim-Se
Fim-Se
Fim-Se

```

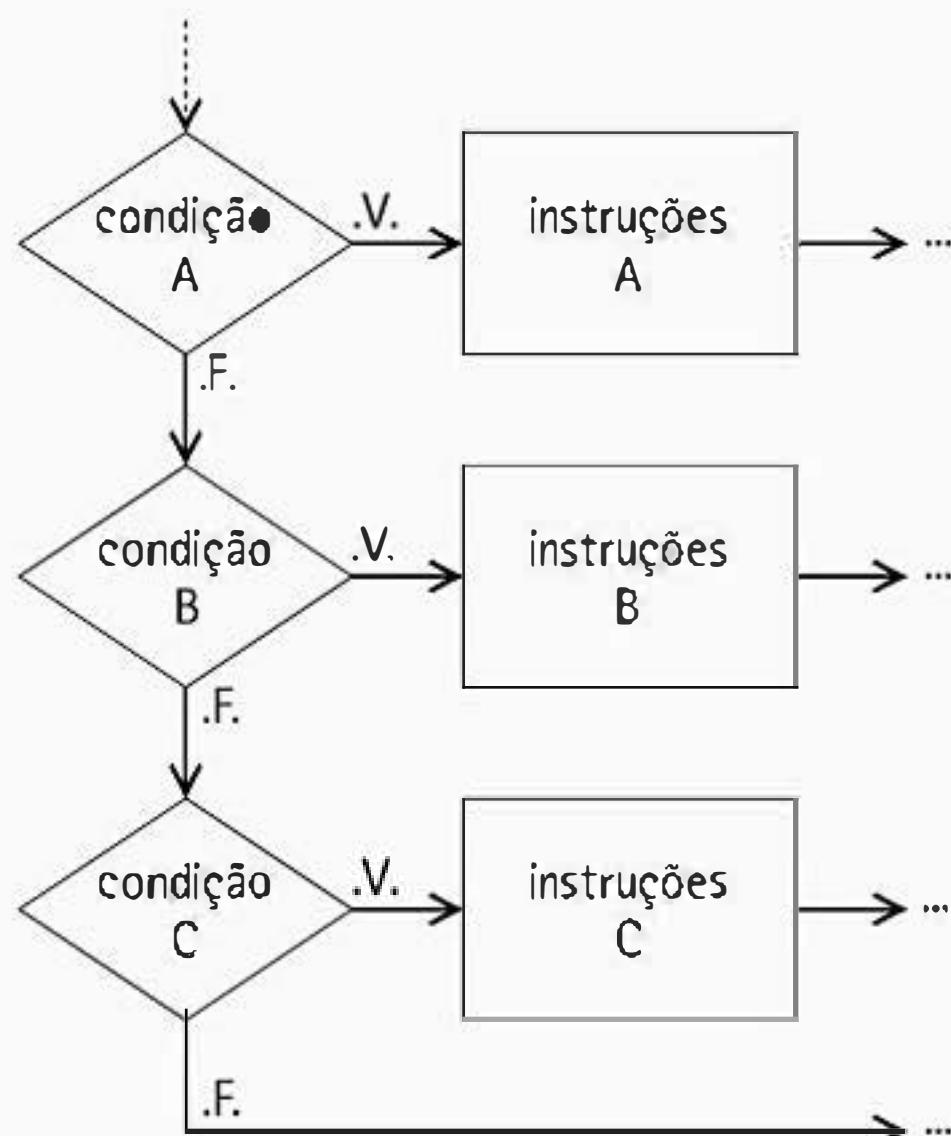
Estrutura com seleção de múltipla escolha

Escolha variável

- Caso Tal_Coisa_1 Faça instrução a
- Caso Tal_Coisa_2 Faça instrução b
- Caso Tal_Coisa_3 Faça instrução c
- Caso Contrário Faça instrução d

Fim-Escolha

Fluxograma:



Java:

```

switch (<variável>) {
    case <Tal_Coisa_1> : <instrução a>;
        [break;]
    case <Tal_Coisa_2> : <instrução b>;
        [break;]
    case <Tal_Coisa_3> : <instrução c>;
        [break;]
    default : <instrução d>
}

```

NOTA:

A palavra reservada `break` é utilizada na linguagem Java para garantir que apenas a instrução selecionada seja executada. Sem esse modificador de fluxo, todas as instruções a partir da seleção encontrada seriam executadas.

EXEMPLO 5.7: Ler o código de um produto e exibir o seu nome de acordo com a tabela a seguir:

Código do produto	Nome do produto
001	Caderno
002	Lápis
003	Borracha
qualquer outro	Diversos

Pseudocódigo utilizando a instrução escolha caso:

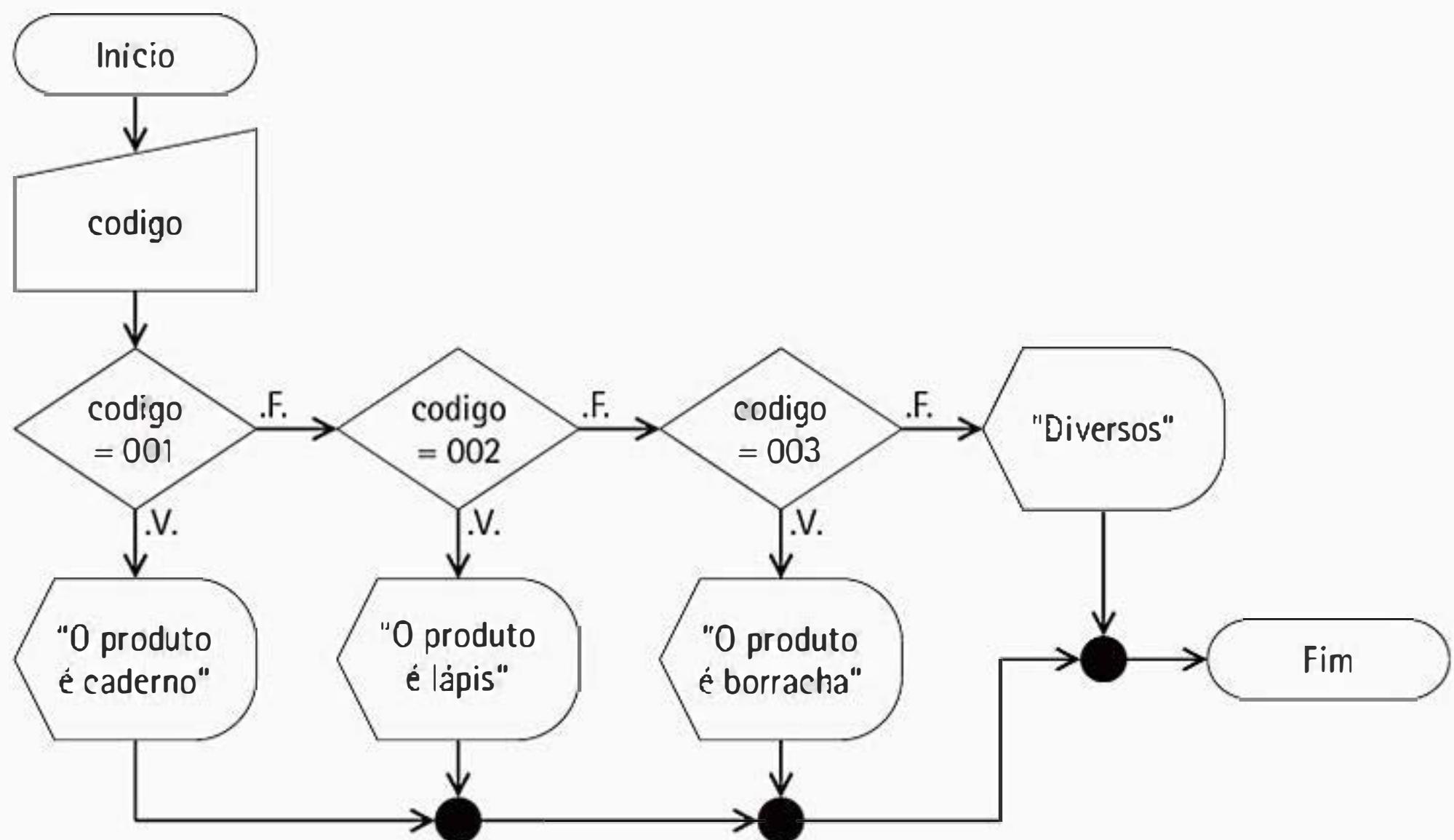
1. Algoritmo produto
2. Var
3. codigo: inteiro
4. Início
5. Ler (codigo)
6. Escolha codigo
7. Caso 001: Mostrar ("O produto é caderno")
8. Caso 002: Mostrar ("O produto é lápis")
9. Caso 003: Mostrar ("O produto é borracha")
10. Caso contrário: Mostrar ("Diversos")
11. Fim-Escolha
12. Fim.

Pseudocódigo utilizando a instrução se:

1. Algoritmo produto
2. Var
3. Codigo: inteiro
4. Início
5. Ler (Código)
6. Se Código = 001 Então
 7. Mostrar ("O produto é caderno")
8. Senão
 9. Se Código = 002 Então
 10. Mostrar ("O produto é lápis")
 11. Senão
 12. Se Código = 003 Então
 13. Mostrar ("O produto é borracha")
 14. Senão
 15. Mostrar("Diversos")
 16. Fim-Se
 17. Fim-Se
 18. Fim-Se
 19. Fim.

Fluxograma:

A representação da resolução por meio do fluxograma é igual para as duas possibilidades, tanto utilizando a instrução escolha quanto a instrução se.

**Java (usando escolha):**

```

1. import java.io.*;
2. class Produto {
3.     public static void main (String args []) {
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
(System.in));
6.         int codigo;
7.         try {
8.             System.out.println("Digite o código: ");
9.             codigo = Integer.parseInt(entrada.readLine());
10.            switch (codigo) {
11.                case 001 : System.out.println ("Caderno");
12.                            break;
13.                case 002 : System.out.println ("Lápis");
14.                            break;
15.                case 003 : System.out.println ("Borracha");
16.                            break;
17.                default : System.out.println ("Diversos");
18.            }
19.        } catch (Exception e) {
20.            System.out.println("Ocorreu um erro durante a
leitura!");
21.        }
22.    }
23. }
  
```

5.8 ESTRUTURAS DE REPETIÇÃO

Em determinadas situações, temos de repetir o programa ou parte dele várias vezes, como no cálculo das médias das notas de um grupo de alunos. Reiniciar o programa para cada cálculo não é uma solução muito prática e algumas vezes é inviável. Uma solução comum é a utilização de **estruturas de repetição**.

O conceito de repetição (ou *looping*) é utilizado quando se deseja repetir um certo trecho de instruções por um número de vezes. O número de repetições pode ser conhecido anteriormente ou não, mas necessariamente precisa ser finito.

Nem todas as estruturas de repetição possuem recursos para fazer a contagem do número de vezes que o laço deverá ser repetido. Por isso, deve-se utilizar uma variável de apoio, sempre do tipo **inteiro**.

Exemplo:

```
Var contador : inteiro
    Inicio
    ...
        contador ← contador + 1
```

NOTA:

Na expressão contador ← contador + 1, o + 1 é denominado incremento. Isso significa que, cada vez que for executado, acrescentará 1 à variável contador. O incremento não precisa ser necessariamente 1, pode ser qualquer outro valor, inclusive negativo; neste caso é denominado decremento.

Às vezes também será necessário acumular valores, isto é, calcular o somatório de um conjunto de valores. Para isso, também será necessário utilizar uma variável de apoio, que pode ser do tipo **inteiro** ou **real**, de acordo com os valores que serão acumulados.

Exemplo:

```
Var
    acumulador: real;
    valor: real;
    Início
    ...
        acumulador ← acumulador + valor; → É a variável que deve ter seu valor
                                         acumulado.
```

5.8.1 ESTRUTURA DE REPETIÇÃO COM TESTE NO INÍCIO – ESTRUTURA ENQUANTO

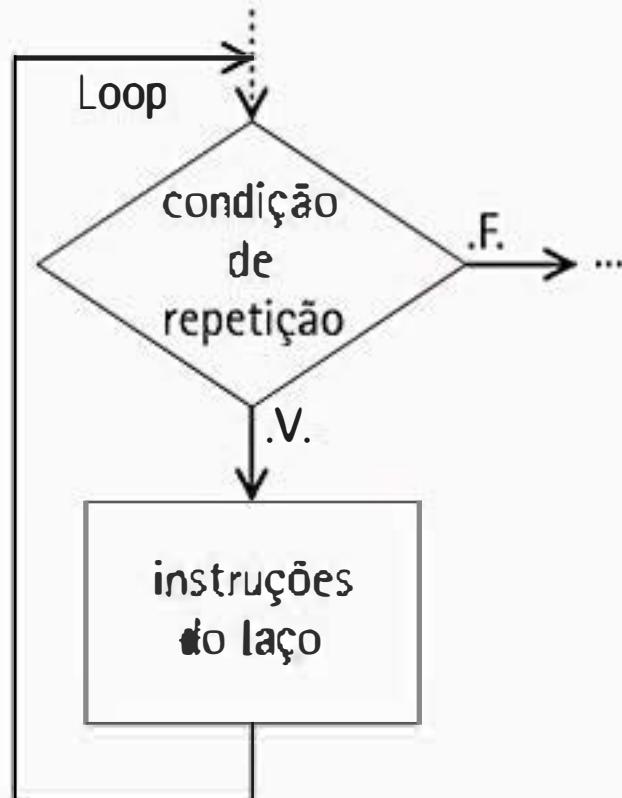
Na estrutura **enquanto**, a condição de repetição é verificada antes de entrar no laço, isto é, uma condição é testada **inicialmente** e, se o resultado for verdadeiro, o bloco de instruções será executado.

NOTA: Laço é o bloco de instruções que serão realizadas repetidas vezes e que estão contidas em uma estrutura de repetição.

Pseudocódigo:

```
Enquanto condição Faça
    <Bloco de instruções>
Fim-Enquanto;
```

Fluxograma:



Java:

```
while (<condição>) {
    <bloco de instruções>
}
```

Conforme esboçado acima, inicialmente é testado o valor da condição. Se ela for verdadeiro, então será executado o comando ou bloco de comandos controlado pela estrutura do `while`. Após essa execução, testa-se novamente o valor da condição. Se a condição ainda for verdadeiro, então os comandos serão executados novamente e assim sucessivamente até que a condição seja falso.

Como o teste da condição é sempre realizado antes da execução dos comandos controlados pela estrutura, se desde o início a condição for falso, então os comandos controlados não serão executados nenhuma vez.

É importante observar que, para terminar a execução do comando `while`, em algum momento os comandos na estrutura `while` devem modificar o valor da condição de controle, de modo que a condição se torne falso. Se isso não acontecer, o processo de repetição ocorrerá indefinidamente e a execução do programa não será terminada.

EXEMPLO 5.8: Ler 850 números fornecidos pelo usuário e calcular e exibir a média.

Pseudocódigo

1. Algoritmo ExEnquanto
2. Var

```

3.      soma, num, media: real
4.      cont: inteiro
5.  Início
6.      soma ← 0
7.      cont ← 0
8.  Enquanto cont < 850 faça
9.    Início
10.       Ler (num)
11.       soma ← soma + num
12.       cont ← cont + 1
13.    Fim
14.    media ← soma / cont
15.    Mostrar ("Média = ", media)
16. Fim.

```

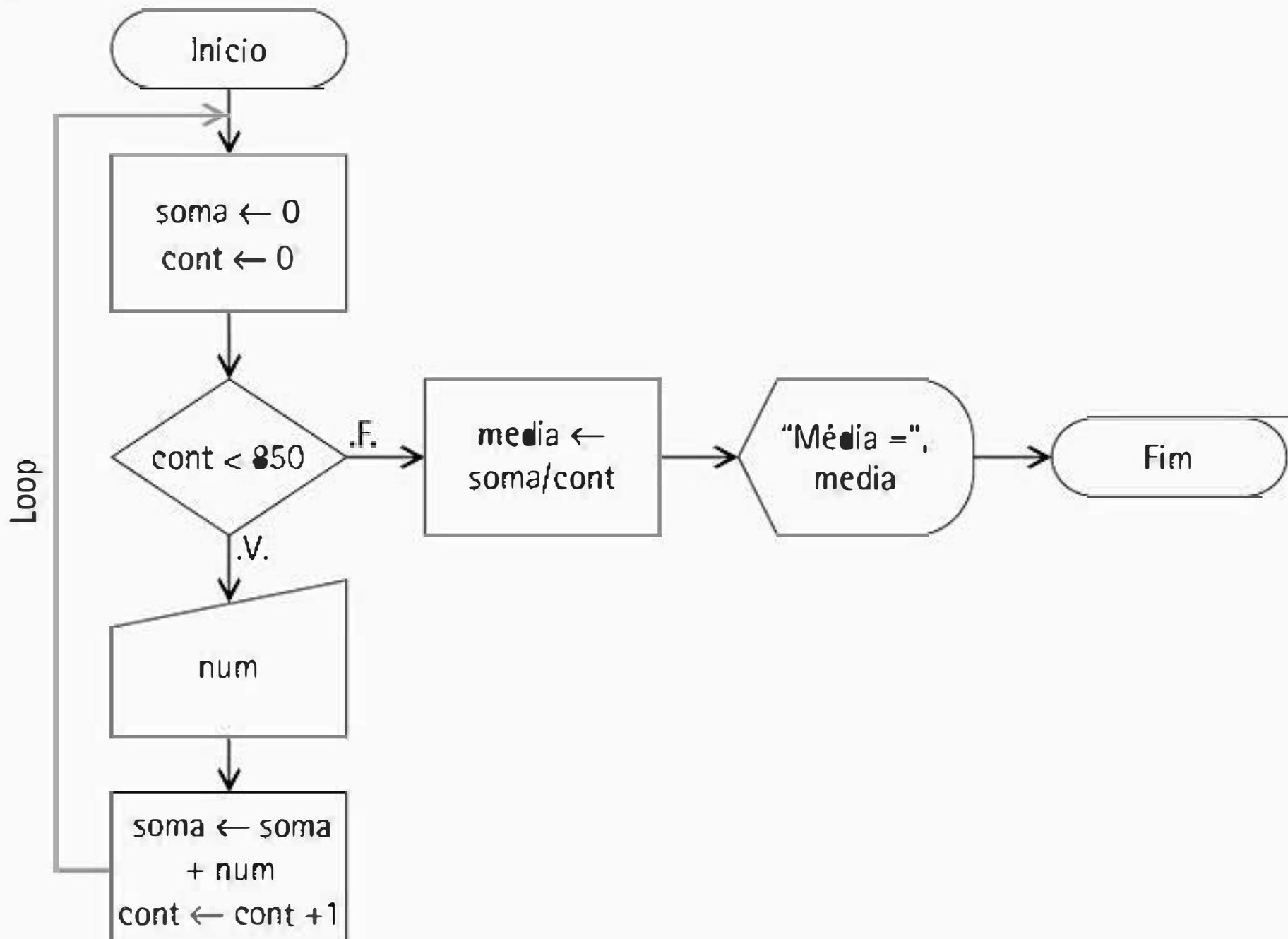
A variável `cont` terá a função de contar o número de vezes que as instruções dentro do laço serão repetidas. A variável `soma` terá a função de acumular todos os valores atribuídos à variável `num`.

O trecho que está entre as linhas 8 e 13 é o conjunto de instruções que será repetido se a condição da linha 8 (`Enquanto cont < 850 faça`) resultar verdadeiro.

A variável `soma` acumulará o seu próprio valor mais o valor de `num` a cada vez que o trecho for executado.

ATENÇÃO: Toda variável que tem a função de contador ou acumulador deve ser inicializada.

Fluxograma:



NOTA: O loop acontece sempre que a condição é satisfeita (verdadeiro).

Java:

```
1. import java.io.*;
2. class ExEnquanto {
3.     public static void main (String args []) {
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
6.             (System.in));
7.         float numero, media, soma;
8.         int cont;
9.         cont = 0;
10.        soma = 0f;
11.        try {
12.            while (cont < 850) {
13.                System.out.println("Digite o número:");
14.                numero = Float.parseFloat(entrada.readLine());
15.                soma = soma + numero;
16.                cont = cont +1;
17.            }
18.            media = soma / cont;
19.            System.out.println("Média = " + media);
20.        } catch (Exception e) {
21.            System.out.println("Ocorreu um erro durante a
22.                            leitura!");
23.        }
24.    }
25. }
```

Sobre o código:

Na linha 9, temos que a soma recebeu o valor 0f. Como já foi discutido, essa é uma particularidade da linguagem Java e a função da letra f é garantir que o número seja considerado do tipo real (float).

Na condição da estrutura de repetição, utilizamos < em vez de <=. Cuidado com esses detalhes. A contagem está começando com 0, mas, quando contamos uma quantidade qualquer, sempre começamos com 1. Portanto, se a condição fosse determinada com <=, estariámos somando um valor a mais. Esse teste pode ser facilmente realizado com quantidades de números menores.

5.8.2 ESTRUTURA DE REPETIÇÃO COM TESTE NO FIM – ESTRUTURA REPITA

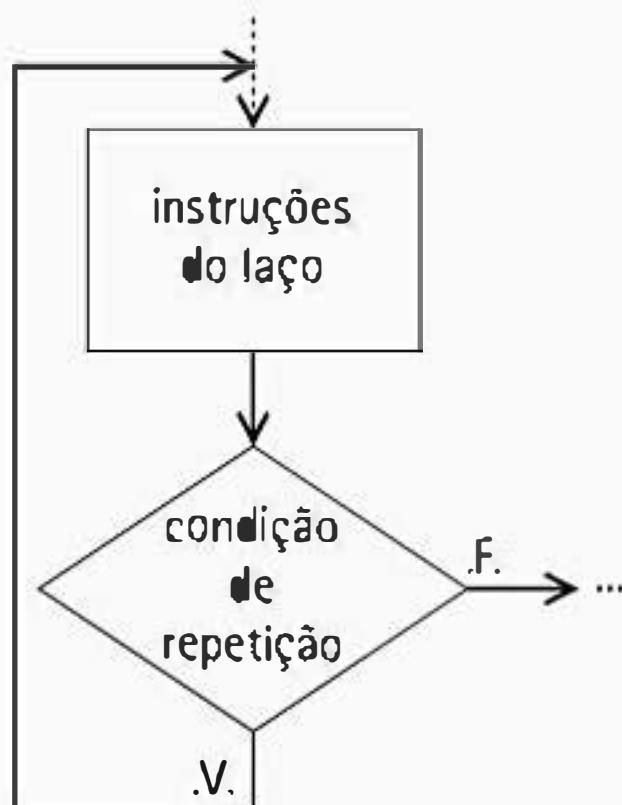
A estrutura de repetição com teste no fim permite a um ou mais comandos serem executados repetidamente até uma condição específica tornar-se verdadeiro. Essa estrutura age de forma muito semelhante à estrutura anterior; a diferença é que os co-

mandos são executados antes de se testar o valor da condição. Como a condição é testada no final, os comandos na estrutura serão executados pelo menos uma vez, desconsiderando-se o valor inicial da condição de controle.

Pseudocódigo:

```
Repita
    <Bloco de instruções>
    Até (<condição>)
```

Fluxograma:



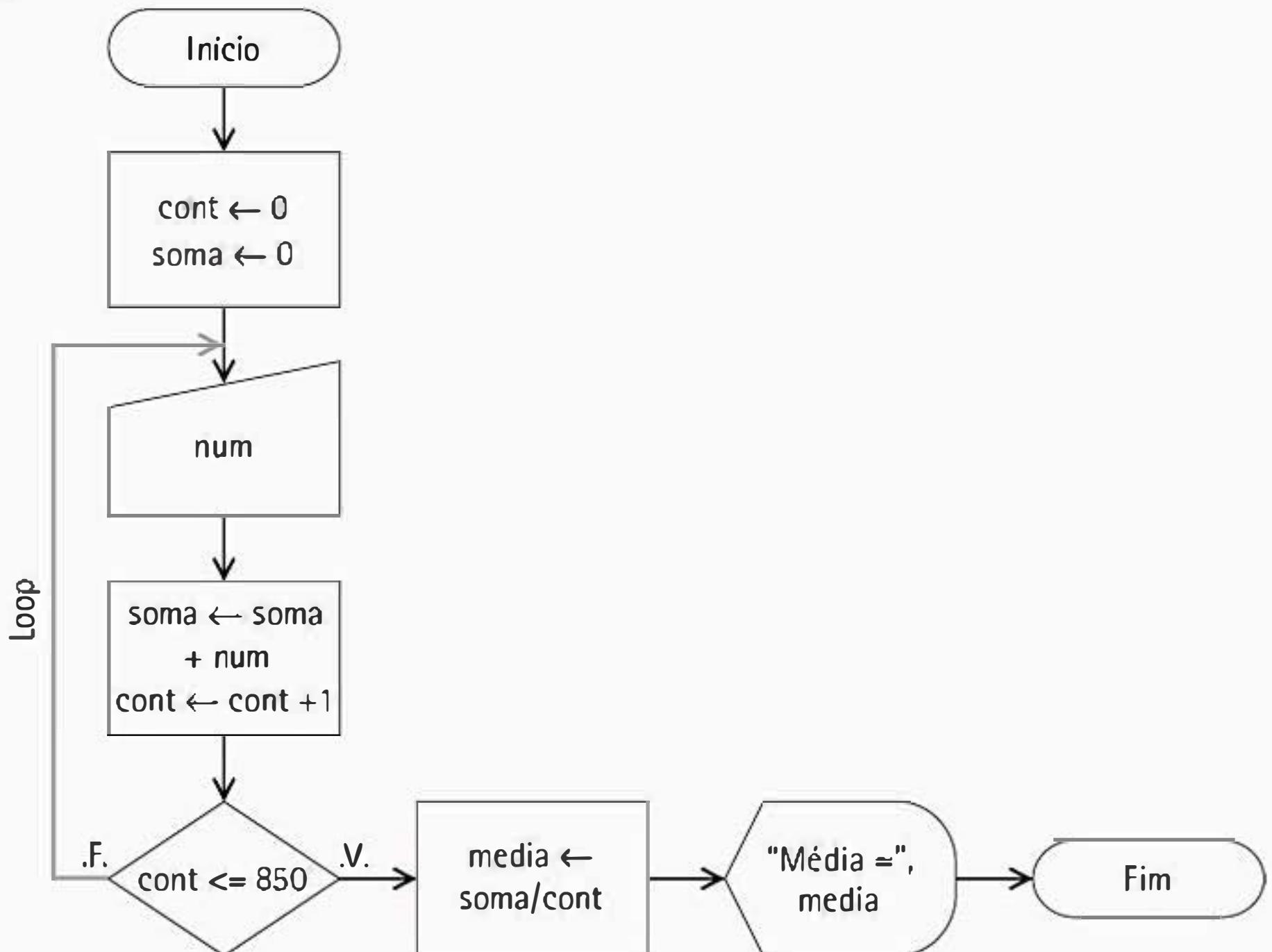
Java:

```
do {
    <bloco de instruções>
} while (<condição>);
```

EXEMPLO 5.9: Ler 850 números fornecidos pelo usuário e calcular e exibir a média.

Pseudocódigo:

1. Algoritmo ex_repita
2. Var
3. soma, num, media: real
4. cont: inteiro
5. Início
6. soma ← 0
7. cont ← 0
8. Repita
9. Ler (num)
10. soma ← soma + no
11. cont ← cont + 1
12. Até que cont => 850
13. media ← soma / cont
14. Mostrar ("Média = ", media)
15. Fim.

Fluxograma:**NOTA:**

O loop acontece até que a instrução seja satisfeita (verdadeiro).

Java:

```

1. import java.io.*;
2. class ExRepita {
3.     public static void main (String args []) {
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
6.             (System.in));
7.         float numero, media, soma;
8.         int cont;
9.         cont = 0;
10.        soma = 0f;
11.        try {
12.            do {
13.                System.out.print("Digite o número:");
14.                numero = Float.parseFloat(entrada.readLine());
15.                soma = soma + numero;
16.                cont = cont +1;
17.            } while (cont < 850);
18.            media = soma / cont;
19.            System.out.println("Média = " + media);
20.        } catch (Exception e) {
21.            System.out.println("Ocorreu um erro durante a
leitura!");
22.        }
23.    }
24. }
    
```

```

21.      }
22.  }
23. }
```

Nos exemplos 5.8 e 5.9 as variáveis de controle que serão testadas a cada repetição (loop) podem ser utilizadas para armazenar números (como no exemplo apresentado) ou letras. Por exemplo, pode-se desejar que um determinado trecho do programa seja repetido de acordo com uma resposta do usuário:

```

Enquanto (resposta = "sim") faça
    <instruções>
        Mostrar ("Deseja continuar?")
        ler (resposta)
Fim enquanto
```

No exemplo anterior é importante padronizar a leitura da resposta, pois sim é diferente de SiM, ou seja, existe diferenciação entre maiúsculas e minúsculas.

5.8.3 ESTRUTURA DE REPETIÇÃO COM VARIÁVEL DE CONTROLE – ESTRUTURA PARA

A estrutura de repetição para utiliza variáveis de controle que definem exatamente o número de vezes que a seqüência de instruções será executada.

Pseudocódigo:

```

Para <var> = <valor Inicial> Até <valor Final> Passo <incremento> Faça
    <Bloco de instruções>
Fim_Para;
```

NOTA: Os argumentos <valor Inicial> e <valor Final> podem ser substituídos por variáveis.

Java:

```

for (<var> = <valor Inicial>; <condição>; <incremento>) {
    <bloco de instruções>
}
```

EXEMPLO 5.10: Ler 850 números fornecidos pelo usuário e calcular e exibir a média.

Pseudocódigo:

1. Algoritmo ex_para
2. Var
3. soma, num, media: real
4. cont: inteiro
5. Início
6. soma ← 0
7. Para cont ← 1 até 850 Passo 1 Faça

```

8.      Ler (no)
9.      soma ← soma + num
10.     Fim-Para
11.     media ← soma / cont
12.     Mostrar ("Média= ", media)
13. Fim.

```

No Exemplo 5.10, tem-se que:

`cont` – variável de controle (= contador);

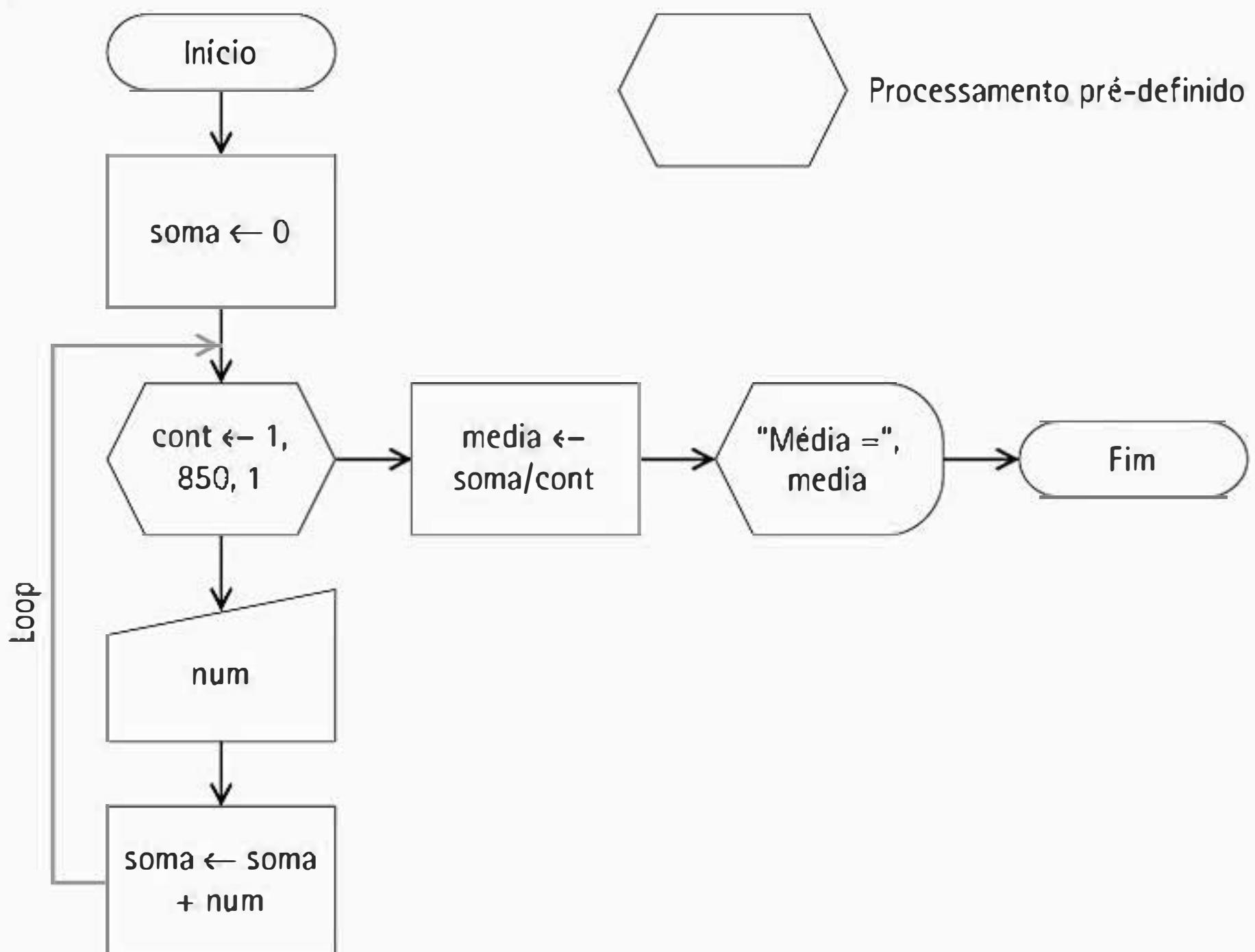
`1` – valor inicial da variável de controle;

`850` – valor final da variável de controle – o contador irá variar de `1` até `850`;

Passo 1 – incremento – representa quanto será acrescido à variável de controle cada vez que o loop acontecer;

substitui a instrução `cont←cont`.

Fluxograma:



Java:

```

1. import java.io.*;
2. class ExPara {
3.     public static void main (String args []) {
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
6.             (System.in));
7.         float numero, media, soma;
8.         int cont;

```

```

8.     soma = 0f;
9.     try {
10.         for (cont = 0; cont < 3; cont++) {
11.             System.out.print("Digite o número: ");
12.             numero = Float.parseFloat(entrada.readLine());
13.             soma = soma + numero;
14.         }
15.         media = soma / cont;
16.         System.out.println("Média = " + media);
17.     } catch (Exception e) {
18.         System.out.println("Ocorreu um erro durante a
leitura!");
19.     }
20. }
21. }
```

Sobre o código:

Na linha 10, o incremento determinado por `cont++` é equivalente a `cont = cont + 1`. Essa sintaxe de compressão de operadores é utilizada com freqüência na linguagem Java.

5.9 EXERCÍCIOS PARA FIXAÇÃO

Para os exercícios 1 e 2, faça o fluxograma e o programa em Java.

1. Ler 300 números divisíveis por 3, calcular a soma entre eles e exibir o resultado.

Estrutura enquanto

Algoritmo Exemplo5_14

var

cont: inteiro
 n, acm: real

Início

cont ← 0, acm ← 0

Enquanto cont ← 300

Ler (n)

Se n mod 3 = 0 Então

cont ← cont + 1

acm ← acm + n

Fim-Se

Fim_Enquanto

Fim.

2. Faça a tabuada de um número qualquer fornecido pelo usuário e multiplicando de 98 até 2.589. Exiba cada resultado.

Estrutura para

Algoritmo Exemplo5_15

var

N, R: real

cont: inteiro

Início

Ler (N)

Para cont:= 98 Até 2589 Passo 1 Faça

R ← N * cont

Mostrar (N, "multiplicado por", cont, "=", R)

Fim-Para

Fim.

5.10 EXERCÍCIOS SUGERIDOS

- Um aluno realizou três provas de uma disciplina. Considerando o critério abaixo, faça um programa que mostre se ele ficou para exame; em caso positivo, qual nota esse aluno precisa tirar no exame para passar.

$$\text{Média} = (\text{Prova1} + \text{Prova2} + \text{Prova3})/3$$

A média deve ser maior ou igual a 7,0. Se não conseguir, a nova média deve ser:

$$\text{Final} = (\text{Média} + \text{Exame})/2$$

Nesse caso, a média final deve ser maior ou igual a 5,0.

- Uma livraria está fazendo uma promoção para pagamento à vista em que o comprador pode escolher entre dois critérios de desconto:

Critério A: R\$ 0,25 por livro + R\$ 7,50 fixo.

Critério B: R\$ 0,50 por livro + R\$ 2,50 fixo.

Faça um programa em que o usuário digita a quantidade de livros que deseja comprar e o programa diz qual é a melhor opção de desconto.

- Dados seis números inteiros representando dois intervalos de tempo (horas, minutos e segundos), faça um programa para calcular a soma desses intervalos. Faça outro programa para calcular a diferença entre os intervalos.
- Faça uma tabela mostrando a tabuada de um número dado pelo usuário.
- Escreva um programa que apresente a tabuada no seguinte formato:

1	2	3	...	n
---	---	---	-----	-----

2	4	6	...	$2n$
---	---	---	-----	------

3	6	9	...	$3n$
---	---	---	-----	------

...

m	$2m$	$3m$...	mn
-----	------	------	-----	------

6. O triângulo de Floyd tem o seguinte formato:

1				
2	3			
4	5	6		
7	8	9	10	
11	12	13	14	15
...				

Isto é, o elemento j da linha i ($j \leq i$) tem o valor $(i-1) * i / 2 + j$. Escreva um programa que gere um triângulo de Floyd com m linhas.

7. Um valor importante nas ciências exatas é o neperiano ou número de Neper ($e = 2,7182$). Além do valor de e , o valor de e^x também é muito importante para cálculos diversos em matemática, economia entre outras e pode ser obtido por meio da série:

$$e^x = 1 + x + \frac{x^2}{2!} + \dots + \frac{x^i}{i!} + \dots$$

cuja relação de recorrência é:

$$e^x = \sum_n t_n = t_1 + t_2 + \dots + t_n$$

onde $t_0 = 1$; $t_n = t_{n-1} * \frac{x}{i}$; $i > 0$

Nessa série, n representa o número de termos e x , o expoente. Faça um programa que calcule e^x , considerando que o usuário entre com os valores de x e n como argumentos.

8. Dado x real que representa um valor em radianos e n natural, calcule uma aproximação para $\cos x$ por meio dos n primeiros termos da seguinte série:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + (-1)^i * \frac{x^{2*i}}{(2*i)!} + \dots$$

Relação de recorrência: $t_i = -t_{i-1} * x^2 / (k_i * (k_i - 1))$, $i > 1$

$$k_i = k_{i-1} + 2$$

$$t_1 = 1 \text{ e } k_1 = 0$$

5.11 EXERCÍCIOS COMPLEMENTARES

1. Escreva um programa que determine o número de dígitos de um valor inteiro maior ou igual a 0.
2. Para converter a temperatura de graus Celsius para Fahrenheit, usa-se a expressão:

$$F = \frac{9}{5} C + 32$$

Faça um programa que efetue essa conversão a partir de uma temperatura dada pelo usuário.

3. Considere a seqüência de Fibonacci, em que o n -ésimo número é dado pela seguinte fórmula de recorrência:

$$F_1 = 1, F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \text{ para } n >= 3$$

ex: $F_1 = 1, F_2 = 1, F_3 = F_2 + F_1 = 1 + 1 = 2, F_4 = F_3 + F_2 = 2 + 1 = 3, \dots$

O quociente de F_n por F_{n-1} converge para o número áureo, isto é, $F_n / F_{n-1} \rightarrow 1.618\dots$. Esse número áureo é usado em arquitetura e outras áreas para determinar relações das dimensões de objetos esteticamente ‘agradáveis’.

- a) Faça um programa que calcule o n -ésimo número de Fibonacci.
- b) Faça um programa que calcule o número áureo com alguma precisão.

Dica:

A convergência é calculada pela soma dos termos dividida pelo número de termos quando esses termos tendem a infinito, isto é,

$$\frac{\sum Q_n}{n} = \frac{Q_1 + Q_2 + \dots + Q_n}{n} \rightarrow 1.618; \quad \text{onde} \quad Q_n = \frac{F_n}{F_{n-1}}$$

4. Considere a seqüência:

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{1}{9.999} - \frac{1}{10.000}$$

Faça um programa para calcular:

- a) A soma de todos os termos de modo direto.
- b) A soma dos termos positivos menos a soma dos termos negativos.

ESTRUTURA DE DADOS: VETORES

- ▶ *Introdução às estruturas de dados*
- ▶ *Conceito de vetores*
- ▶ *Conceito de matrizes*
- ▶ *Exemplos e exercícios para fixação*

OBJETIVOS:

Estudar as estruturas de dados estáticas e homogêneas (vetores e matrizes), as operações que estas suportam e algumas aplicações básicas.

Até agora vimos instruções básicas para realizar seqüências lógicas de instruções que permitem atingir um resultado desejado. Dessa forma, trabalhamos com valores simples, definidos ou determinados pelo usuário durante a operação do algoritmo e armazenados em uma variável de memória.

Esse armazenamento de variáveis na memória tem sido, até o momento, suficiente para nossas necessidades, porém existem casos em que necessitamos armazenar, não um único valor, mas um conjunto de valores. Suponha, por exemplo, o caso de um treino de classificação de uma corrida de Fórmula 1, em que é necessário verificar os tempos obtidos por todos os pilotos para avaliar qual será o primeiro no grid de largada. Para fazer essa ordenação, é necessário armazenar o tempo de todos os pilotos e, depois, realizar a ordenação desses tempos.

Com o objetivo de tratar conjuntos de dados, podem-se armazenar os valores em disco, por meio de arquivos, como veremos no Capítulo 8, ou em variáveis de memória. Nesse segundo caso, pode-se criar uma variável para cada valor ou criar uma estrutura que permita armazenar um conjunto de valores de forma que possamos acessar o valor ou valores desejados.

6.1 ESTRUTURAS INDEXADAS – VETOR (ARRAY)

Nos casos em que é necessário ou conveniente representar os dados em termos de conjuntos de valores no lugar da utilização de variáveis armazenando valores de forma isolada, costumam-se utilizar estruturas de dados especiais denominadas **estruturas indexadas**.

Nesse tipo de estrutura, diversos valores são armazenados em uma estrutura de dados mais complexa cujos elementos individuais são identificados com o auxílio de índices. O exemplo mais simples desse tipo de estrutura é definido como uma estrutura indexada simples (unidimensional) de dados de mesmo tipo. Essa estrutura, que necessita apenas de um índice para identificar um determinado elemento armazenado nela, é chamada, normalmente, de **vetor**.

Um vetor é representado como uma linha de contêineres de valores identificados por índices.

índice:	0	1	2	3	4	5	6	7	8	9
nome:										

Assim, um vetor é uma coleção de variáveis de um mesmo tipo que compartilham o mesmo nome e que ocupam posições consecutivas de memória. Cada variável da coleção denomina-se **elemento** e é identificada por um **índice**.

Para manipularmos um determinado valor (elemento) em um vetor, precisamos fornecer o nome do vetor (identificador) e o índice do elemento desejado. Esse índice determina a posição em que o elemento está inserido na estrutura. Cada posição do vetor contém exatamente um valor que pode ser manipulado individualmente. Naturalmente, a dimensão (tamanho) e os índices que indicam o elemento selecionado são definidos por números inteiros.

6.1.1 DECLARAÇÃO DE VETOR

Um vetor é declarado definindo-se seu nome, que é um identificador válido da linguagem, seu tipo, que define o tipo individual de cada elemento do vetor, e seu tamanho, que determina quantos valores o vetor poderá armazenar. De modo geral, utilizam-se os colchetes para declarar um vetor e identificar um elemento específico desse vetor.

O significado dos valores armazenados no vetor depende da aplicação em que a estrutura for usada.

Representação da declaração de um vetor em pseudocódigo:

```
V : vetor [0..N] de inteiros
```

Essa declaração define uma variável chamada V que pode armazenar um conjunto de números inteiros que serão identificados como $V[0]$, $V[1]$, $V[2]$, ..., $V[N]$, isto é, temos um conjunto de números inteiros, cada qual em um endereço seqüencial diferente definido como o índice do vetor. Assim, $V[0]$ guarda o primeiro número inteiro, $V[1]$ guarda o segundo número e sucessivamente até $V[N]$, que contém o último número armazenado. De forma geral, $V[i]$ (lê-se ‘V índice i’) guarda o i -ésimo elemento do vetor V. Supondo que esse vetor tenha dez elementos e receba os seguintes valores [58, 4, 0, 123, 8, 59, 1, 500, ...N] temos:

índice	0	1	2	3	4	5	6	7	8	9
valor	58	4	0	123	8	59	1	500	758	2

Sendo assim,

elemento	1º	2º	3º	4º	5º	6º	7º	8º	9º	10º
variável	$V[0]$	$V[1]$	$V[2]$	$V[3]$	$V[4]$	$V[5]$	$V[6]$	$V[7]$	$V[8]$	$V[9]$
valor	58	4	0	123	8	59	1	500	758	2

Observe que, como o primeiro índice do vetor é 0, a quantidade de elementos guardada em um vetor é dada pelo maior índice mais 1, isto é, um vetor que varia de 0 a 9 tem 10 elementos.

NOTA:

Algumas linguagens, como a Pascal, permitem determinar o primeiro índice do vetor, mas, em Java, o primeiro índice é sempre 0!

Em Java, a declaração do vetor se dá definindo-se o tipo e um identificador e acrescentando-se colchetes.

```
<tipo> <identificador> [ ];
ou
<tipo> [ ] <identificador>;
```

Em Java, os vetores são objetos, permitindo o uso de atributos (propriedades) e a aplicação de métodos. Com isso, além da declaração, é necessário criar esse objeto na memória, determinando seu tamanho, para poder utilizá-lo. Essa criação pode ser feita utilizando-se o operador de criação new.

```
<tipo> [] <identificador> = new <tipo> [n];
ou
<tipo> <identificador> [] = new <tipo> [n];
```

Exemplo:

```
int [] vetor = new int [n];
ou
int vetor [] = new int [n];
```

As declarações acima são equivalentes e definem um vetor de números inteiros que pode armazenar n valores inteiros. Como, em Java, todas as estruturas indexadas têm índice 0 para seu primeiro elemento, é necessário apenas indicar a quantidade de elementos.

6.1.2 ACESSO E ATRIBUIÇÃO DE VALOR EM VETORES

Uma vez criado um vetor, a atribuição de valores é processada de elemento em elemento, alterando-se o valor do índice do vetor.

```
v : vetor [0..N] de inteiros
v[0] ← <valor0>
v[1] ← <valor1>
...
v[N] ← <valorN>
```

Exemplo: Um vetor usado para guardar os nomes dos meses do ano poderia ser declarado:

Pseudocódigo:

```
Meses : vetor [1..12] de inteiros
Meses [1] ← "janeiro"
Meses [2] ← "fevereiro"
...
Meses [11] ← "novembro"
Meses [12] ← "dezembro"
```

Em geral, um vetor pode ser indexado com qualquer expressão cujo valor seja um número inteiro. Essa expressão pode ser uma simples constante, uma variável ou então uma expressão contendo operadores aritméticos, constantes e variáveis.

Java possui tratamentos diferenciados para tipos de dados primitivos e para tipos de dados objetos. Até o momento, não precisamos nos preocupar com a diferença, pois apenas trabalhamos com tipos primitivos como `int`, `char` etc. Ao trabalharmos com vetores, não podemos mais ignorar o tratamento com objetos, pois o significado dos valores armazenados nesses dois tipos é diferente.

Nas variáveis declaradas como tipos primitivos, são guardados valores desses tipos, enquanto nos objetos são guardadas referências aos valores e não os valores propriamente ditos. Vetores, em Java, são objetos e, portanto, seus índices representam referências aos objetos construídos pelo operador new e não aos valores em si. Dessa forma, um vetor não precisa ter seu tamanho previamente definido pelo programador, podendo ser calculado pelo programa em tempo de execução.

```
int tamanho = ... ; // algum cálculo para tamanho
int [] n = new int [tamanho];
```

O valor do tamanho especificado quando o vetor é criado pode ser calculado por qualquer expressão que retorne um valor inteiro positivo. Um vetor pode ser de qualquer tamanho finito, mas deve ser lembrado que cada elemento usará um espaço de memória, o que significa que vetores definidos com tamanho muito grande provavelmente afetarão a eficiência do programa. Se o vetor for grande demais para a manipulação pelo computador, isso poderá provocar uma falha de programa, dependendo do sistema utilizado.

```
<tipo> [ ] <identificador> = new <tipo> [n];
<identificador>[0] = <valor0>;
<identificador>[1] = <valor1>;
...
<identificador>[n-1] = <valorN>;
```

CUIDADO!

Como o primeiro índice é 0, um vetor com tamanho N tem elementos que variam de [0] a [N - 1]. Por exemplo, um vetor com 10 elementos tem um índice que varia de 0 a 9.

A leitura, ou acesso, desses valores é executada da mesma forma, utilizando-se os índices para definir o elemento desejado.

Exemplo:

```
String mesAno [] = new String [12];

mesAno[0] = "janeiro";
mesAno[1] = "fevereiro";
...
mesAno[10] = "novembro";
mesAno[11] = "dezembro";

String mesAniversario = mesAno[0];
```

Um atrativo dos vetores é que sua indexação permite o acesso a qualquer elemento em qualquer instante e em qualquer ordem, sem que sua posição no vetor impõe qualquer custo extra de eficiência. Freqüentemente, têm-se operações aplicadas a

conjuntos de elementos ou mesmo a todos os elementos de um vetor. Nesses casos, é comum utilizar uma estrutura de repetição para varrer os índices desejados. Por exemplo, a estrutura a seguir soma todos os números entre os índices 1 e 6 do vetor num dado.

Pseudocódigo:

```
Algoritmo exemplo_vetor
var
    num: vetor [1..6] de inteiros
    soma : inteiro
    i : inteiro
início
    soma ← 0
    Para i ← 1 até 6 Faça
        soma ← soma + num[i]
    Fim-Para
```

Deve-se ter uma atenção especial com os limites de um vetor. O contador, na estrutura de repetição, deve estar limitado à quantidade de elementos do vetor. No caso acima, para que não haja erros, o tamanho máximo permitido para vetor é 6 (índices que variam de 1 a 6).

Java:

```
Integer num [] = new Integer [6];
int soma = 0;

for (int i = 0; i < 6; i++){
    soma = soma + num[i];
}
```

Nesse caso, como estamos trabalhando com a linguagem Java, os valores considerados para num variam de 0 a 6, uma vez que, conforme foi dito anteriormente, os vetores sempre se iniciam a partir de 0.

CUIDADO!

Uma tentativa de acessar um elemento fora dos limites do vetor resultará em erro de execução. Em Java, esse erro será assinalado com a mensagem: `ArrayIndexOutOfBoundsException`.

Para as operações que envolvem todos os elementos do vetor, em Java, podem-se reduzir as chances de erro fazendo uso do fato de o vetor ser um objeto. Como um vetor sempre conhece seu próprio tamanho, pode-se fazer uso da propriedade `length`, definida pela linguagem para qualquer vetor e que retorna a quantidade de elementos arma-

zenados no vetor. Dessa forma, a estrutura de repetição acima pode ser reescrita como segue:

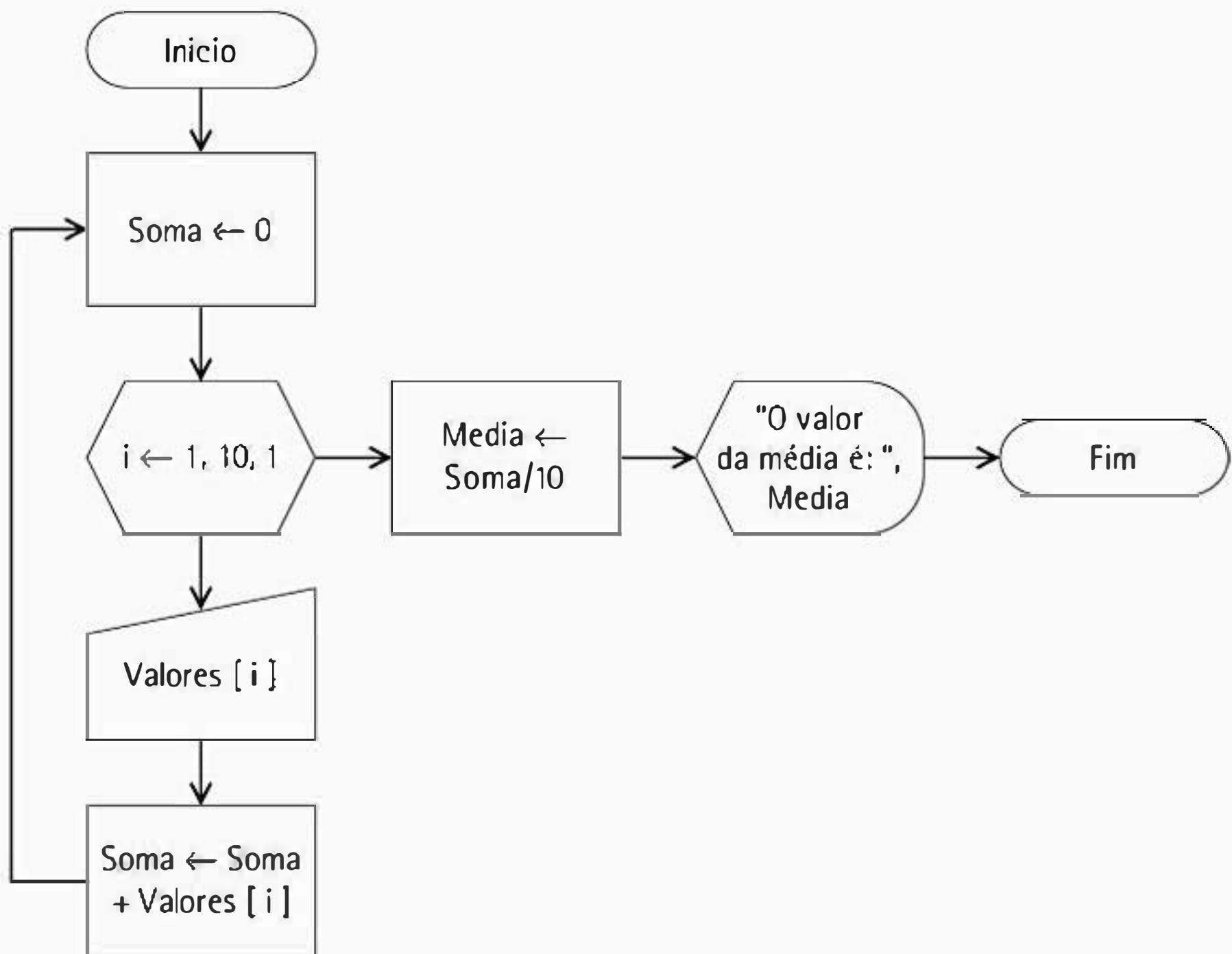
```
int soma = 0;
for (int i = 0; i < num.length; i++) {
    soma = soma + num[i];
}
```

Nessa estrutura, a repetição será realizada desde o primeiro até o último termo (`num.length`) do vetor `num`. Essa solução é, sem dúvida, mais adequada para a solução de problemas que envolvam o vetor como um todo. Suponha que, no futuro, o vetor `num` mude de tamanho: essa segunda expressão continuaria válida, enquanto a primeira teria de ser localizada e alterada e o código teria de ser recompilado.

EXEMPLO 6.1: O programa a seguir faz a leitura de dez valores em um vetor e apresenta o valor da média aritmética desses valores.

Pseudocódigo:

1. Algoritmo Exemplo6.1
2. Var
3. Valores : vetor [1..10] de reais
4. Soma, Media : real
5. i : inteiro
6. Início
7. Soma ← 0
8. Para i ← 1 até 10 Faça
9. Ler (Valores[i])
10. Soma ← Soma + Valores[i]
11. Fim-Para
12. Media ← Soma/10
13. Mostrar ("O valor da média é: ", Media)
14. Fim.

Fluxograma:**Java:**

```

1. import java.io.*;
2. class ExVetor {
3.     public static void main (String args []) {
4.         BufferedReader entrada;
5.         entrada = new BufferedReader(new InputStreamReader
6.             (System.in));
7.         try {
8.             float soma = 0;
9.             float [] vetor = new float [10];
10.            for (int i = 0, i < vetor.length; i++) {
11.                System.out.println("Qual o valor?");
12.                vetor [i] = Float.parseFloat(entrada.readLine());
13.                soma = soma + vetor[i];
14.            }
15.            float media = soma / vetor.length;
16.            System.out.println ("A média = " + media);
17.        } catch (Exception e) {
18.            System.out.println("Ocorreu um erro durante a
19.                leitura!");
20.        }
  
```

6.1.3 OPERAÇÕES

Os vetores permitem a manipulação dos elementos alocados em suas posições de forma independente, como vimos anteriormente. Assim, é possível realizar operações com seus elementos, como o cálculo da média apresentado no Exemplo 6.1.

Vale lembrar que a inclusão de um elemento em uma determinada posição de um vetor implica a substituição do valor anteriormente existente, de forma que a inclusão e substituição de valores consistem em uma única operação.

A exibição dos elementos pode ser necessária após a execução de uma determinada operação que envolva a alteração dos valores desses elementos, de modo que o usuário pode verificar a ocorrência do efeito desejado:

```

Para i ← 1 até 6 Faça
    Mostrar (Valores[i])
Fim-Para

```

EXEMPLO 6.2: • objetivo deste exemplo é desenvolver um algoritmo que efetue a leitura de dez elementos inteiros de um vetor `Teste1`. Vamos construir um vetor `Teste2` do mesmo tipo, observando a seguinte regra de formação: se o valor do índice for par, o valor do elemento deverá ser multiplicado por 5; se for ímpar, deverá ser somado com 5. Ao final, mostrar o conteúdo dos dois vetores.

Pseudocódigo:

```

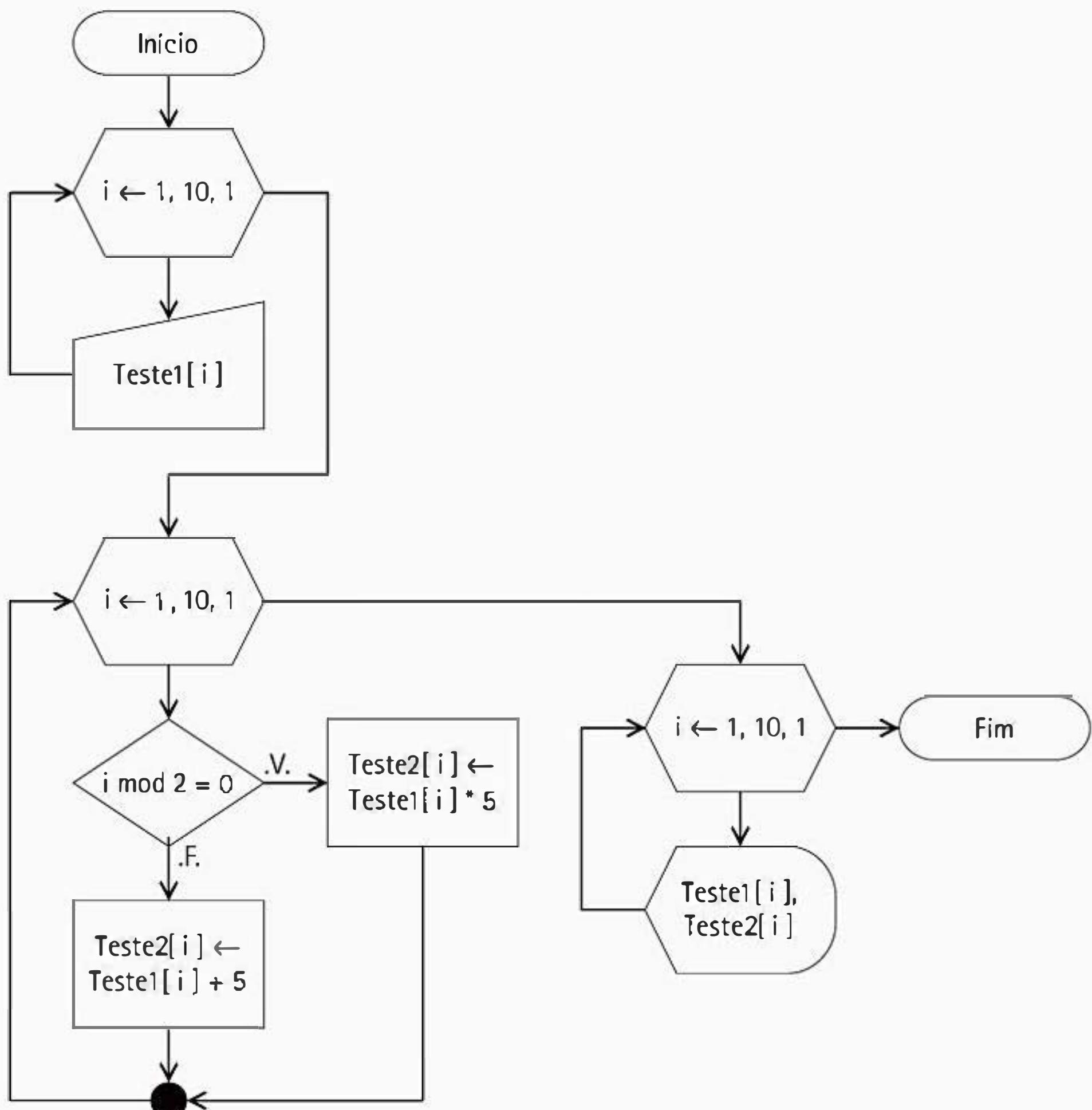
1. Algoritmo Exemplo6.2
2. Var
3.     Teste1, Teste2 : vetor [1..10] de inteiros
4.     i : inteiro
5. Início
6.     Para i ← 1 até 10 Faça
7.         Ler (Teste1[i])
8.     Fim-Para
9.     Para i ← 1 até 10 Faça
10.        Se (i mod 2 = 0) Então
11.            Teste2[i] ← Teste1[i] * 5
12.        Senão
13.            Teste2[i] ← Teste1[i] + 5
14.        Fim-Se
15.    Fim-Para
16.    Para i ← 1 até 10 Faça
17.        Mostrar (Teste1[i], Teste2[i])
18.    Fim-Para
19. Fim.

```

Nesse exemplo é utilizado o operador `mod`, que retorna o resto da divisão de um número por outro. Esse recurso foi utilizado para o teste que verifica se o índice do vetor

é par ou ímpar. Ou seja, ao fazer a divisão de um número por 2, se o resto da operação resulta em zero, o número é par. Na expressão $\text{Se } i \bmod 2 = 0$ Então é realizada essa verificação. Assim, identifica-se o tipo do índice (par ou ímpar) e, dependendo do resultado, executa-se a operação solicitada, atribuindo-se o resultado ao respectivo elemento do **Vetor2**.

Fluxograma:



Java:

```

1. import java.io.*;
2.
3. class Exemplo62 {
4.
5.     public static void main (String args []) {
6.         int teste1 [] = new int [10];
7.         int teste2 [] = new int [10];
8.         BufferedReader entrada;
  
```

```

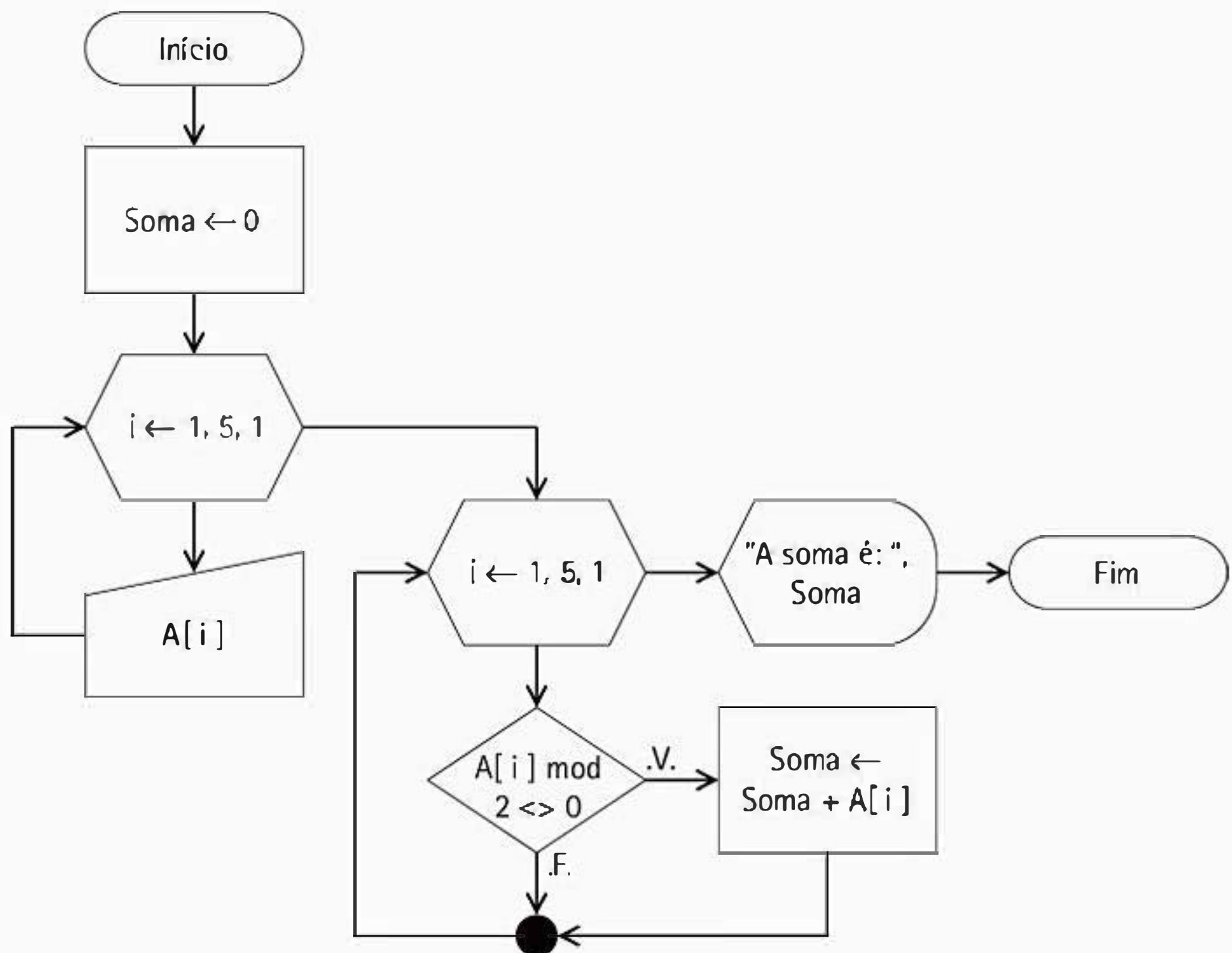
9.         entrada = new BufferedReader(new InputStreamReader
10.            (System.in));
11.        try {
12.            for (int i = 0; i < 10; i++) {
13.                System.out.println("Qual o número?");
14.                teste1[i] = Integer.parseInt(entrada.readLine());
15.            }
16.            for (int i = 0; i < 10; i++){
17.                if (i % 2 == 0)
18.                    teste2[i] = teste1[i] * 5;
19.                else
20.                    teste2[i] = teste1[i] + 5;
21.            }
22.            System.out.println();
23.            System.out.println("Resultado:");
24.            for (int i = 0; i < 10; i++){
25.                System.out.print("teste1[" + i + "] = " + teste1[i] +
26.                  "\t");
27.                System.out.println("teste2[" + i + "] = " +
28.                  teste2[i]);
29.            }
30.        } catch (Exception e) {
31.            System.out.println("Ocorreu um erro durante a leitura!");
32.        }

```

EXEMPLO 6.3: Desenvolver um algoritmo que efetue a leitura de cinco elementos para um vetor A. No final, apresentar a soma de todos os elementos que sejam ímpares.

Pseudocódigo:

1. Algoritmo Exemplo6.3
2. Var
3. Soma, i : inteiro
4. A : vetor[1..5] de inteiros
5. Início
6. Soma ← 0
7. Para i ← 1 até 5 Faça
8. Ler (A[i])
9. Fim-Para
10. Para i ← 1 até 5 Faça
11. Se (A[i] mod 2) <> 0 Então
12. Soma ← Soma + A[i]
13. Fim-Se
14. Fim-Para
15. Mostrar ("A soma é: ", Soma)
16. Fim.

Fluxograma:**Java:**

```

1. import java.io.*;
2.
3. class Exemplo63 {
4.
5.     public static void main (String args []) {
6.         final int tamanho = 5;
7.         int vetor [] = new int [tamanho];
8.         BufferedReader entrada;
9.         entrada = new BufferedReader(new InputStreamReader
10.           (System.in));
11.         try {
12.             for (int i = 0; i < tamanho; i++) {
13.                 System.out.println("Qual o numero?");
14.                 vetor[i] = Integer.parseInt(entrada.readLine());
15.             }
16.             int soma = 0;
17.             for (int i = 0; i < tamanho; i++){
18.                 if (vetor[i] % 2 != 0)
19.                     soma = soma + vetor[i];
20.             }
21.             System.out.println();
22.             System.out.println("Soma = " + soma);
23.         } catch (Exception e) {
  
```

```

23.         System.out.println("Ocorreu um erro durante a leitura!");
24.     }
25. }
26. }
```

Na linha 6 foi declarada uma variável, `tamanho`, caracterizada como `final int`. Essa declaração determina que esse elemento é do tipo inteiro e não se altera ao longo do programa, isto é, ele é uma constante.

Quando temos um valor que será utilizado várias vezes, é mais conveniente declarar esse valor como constante, de forma que alterações sobre esse valor só necessitam ser feitas em um único ponto do programa. Como exercício, experimente realizar essa modificação no programa `Exemplo6.2`.

Freqüentemente, os vetores são utilizados para operação com dados agregados ou que representam um conjunto de elementos que têm uma relação entre si. Muitas aplicações poderiam empregá-los. Uma delas poderia ser o tratamento de dados estatísticos.

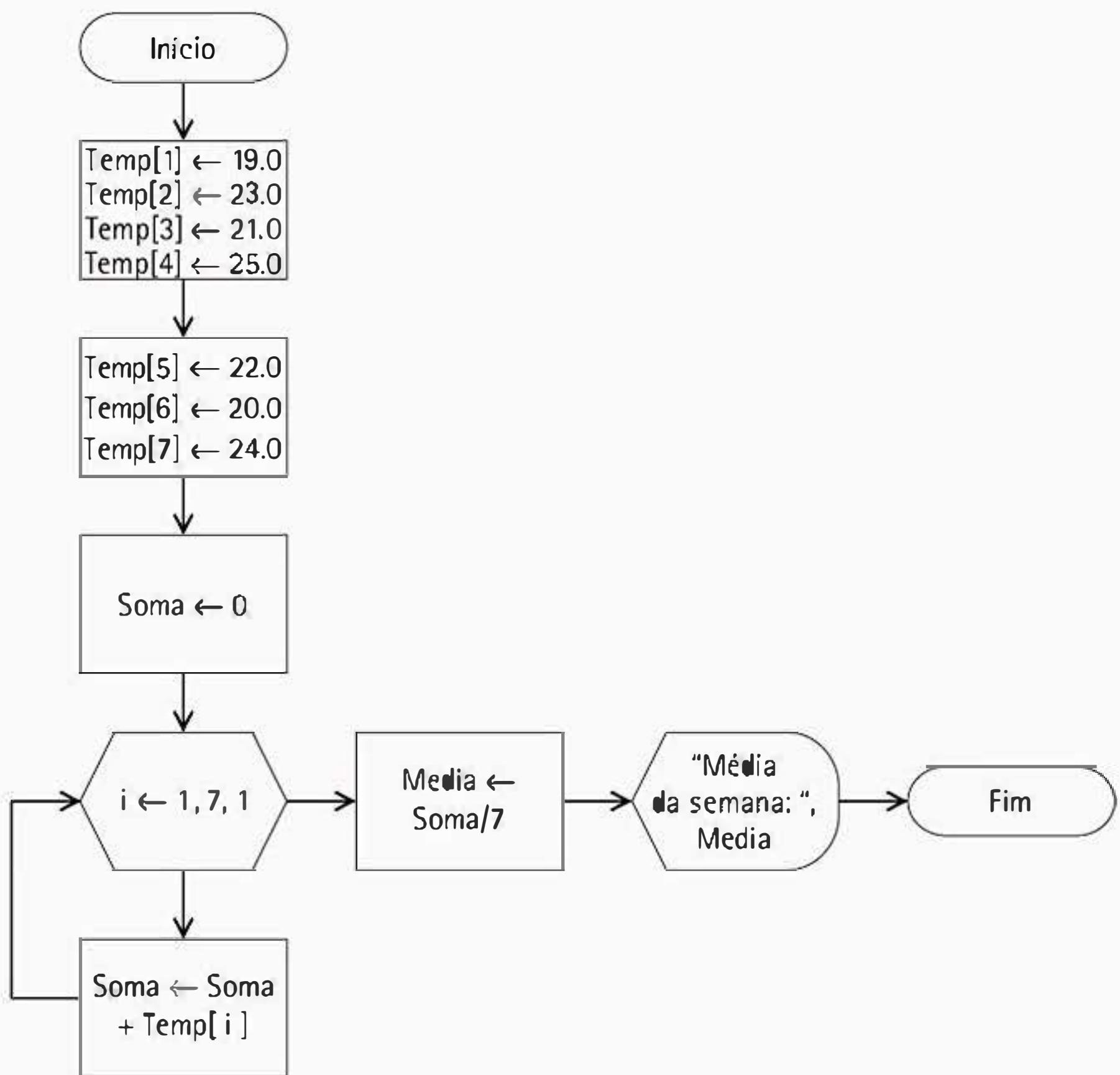
Supondo que se queira guardar as informações referentes às médias diárias das temperaturas verificadas no decorrer de uma semana e executar algumas operações simples, como calcular a temperatura média da semana, classificar essas médias em ordem crescente ou exibir a menor e a maior delas, poderia ser utilizada uma estrutura do tipo vetor, conforme exemplo a seguir.

EXEMPLO 6.4: Este exemplo calcula a média das temperaturas verificadas durante a semana a partir das médias diárias já obtidas.

Pseudocódigo:

```

1. Algoritmo Exemplo6.4
2. Var
3.     Temp : vetor [1..7] de reais
4.     Soma, Média : real
5.     i : inteiro
6. Início
7.     Temp[1] ← 19.0
8.     Temp[2] ← 23.0
9.     Temp[3] ← 21.0
10.    Temp[4] ← 25.0
11.    Temp[5] ← 22.0
12.    Temp[6] ← 20.0
13.    Temp[7] ← 24.0
14.    Soma ← 0
15.    Para i ← 1 até 7 Faça
16.        Soma ← Soma + Temp[i]
17.    Fim
18.    Média ← Soma / 7
19.    Mostrar ("Média da semana: ", Média)
20. Fim.
```

Fluxograma:**Java:**

```

1. class Exemplo64 {
2.
3.     public static void main (String args []){
4.         final int diasSemana = 7;
5.         float temperatura [] = new float [diasSemana];
6.         temperatura [0] = 19.0f;
7.         temperatura [1] = 23.0f;
8.         temperatura [2] = 21.0f;
9.         temperatura [3] = 25.0f;
10.        temperatura [4] = 22.0f;
11.        temperatura [5] = 20.0f;
12.        temperatura [6] = 24.0f;
13.        float soma = 0f, media;
14.        for (int i = 0; i < diasSemana; i++){
15.            soma = soma + temperatura[i];
16.        }
17.        media = soma/diasSemana;
18.        System.out.println();
19.        System.out.println("Média da Semana = " + media);
    
```

```
20. }
21. }
```

EXEMPLO 6.5: Este exemplo efetua a ordenação dos elementos considerados no exemplo anterior, exibindo o maior e o menor deles.

Pseudocódigo:

```

1. Algoritmo Exemplo6.5
2. Var
3.     Temp : vetor [1..7] de reais
4.     x : real
5.     i, j, min : inteiro
6. Início
7.     Temp[1] ← 19.0
8.     Temp[2] ← 23.0
9.     Temp[3] ← 21.0
10.    Temp[4] ← 25.0
11.    Temp[5] ← 22.0
12.    Temp[6] ← 20.0
13.    Temp[7] ← 24.0
14.    Para i ← 1 até 6 Faça
15.        min ← i
16.        Para j ← i + 1 até 7 Faça
17.            Se Temp[j] < Temp[min] Então
18.                min ← j
19.                x ← Temp[min]
20.                Temp[min] ← Temp[i]
21.                Temp[i] ← x
22.            Fim-Se
23.        Fim-Para
24.    Fim-para
25.    Mostrar ("Mínimo: ", Temp[1])
26.    Mostrar ("Máximo: ", Temp[7])
27. Fim.
```

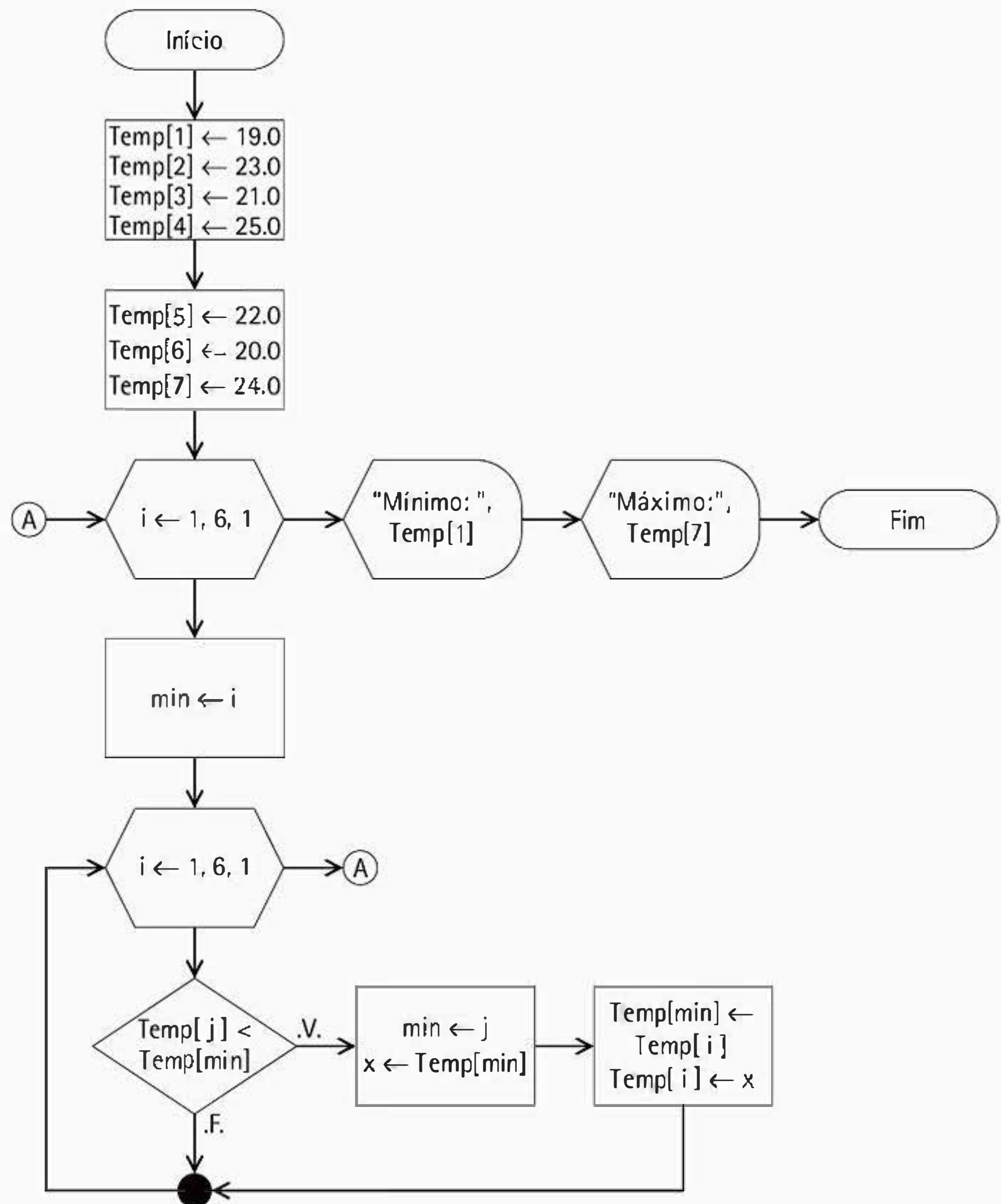
Nesse exemplo, utilizou-se um algoritmo de ordenação para obter os resultados desejados. Métodos de ordenação são importantes em grande parte da computação, por isso veremos os princípios desses algoritmos mais adiante, no Capítulo 9.

A estratégia desse algoritmo é percorrer o vetor do primeiro ao penúltimo elemento. Isso é feito por meio da instrução `para` da linha 14. Nesse percurso, é feita a comparação entre a posição corrente e os elementos das posições subsequentes, por meio da instrução da linha 16. A comparação efetivamente ocorre na linha 17, por meio do comando `se`. Dependendo do resultado dessa comparação, são executados os comandos que fazem a troca de posição dos elementos (linhas 19 a 21). Esse processo se repete até que a penúltima posição seja comparada com a última, não havendo, portanto, necessidade de continuidade do processo.

A tabela a seguir apresenta o teste de mesa para demonstrar, em cada momento da execução, os valores que cada uma das variáveis envolvidas está assumindo. É possível, também, observar-se o resultado final.

Note que i varia de 1 até 6, enquanto j varia de $i + 1$ até 7. Ou seja, quando i tem valor 2, j varia de 3 até 7. Quando o resultado da comparação $\text{Se Temp}[j] < \text{Temp}[\text{min}] \text{ Então}$ (linha 17) é falso, as variáveis min e x mantêm seus valores anteriores, uma vez que não ocorre a execução do código das linhas 18 a 21.

Temp[1]	Temp[2]	Temp[3]	Temp[4]	Temp[5]	Temp[6]	Temp[7]	i	j	min	x
19.0	23.0	21.0	25.0	22.0	20.0	24.0	1	2		
19.0	23.0	21.0	25.0	22.0	20.0	24.0	1	3		
19.0	23.0	21.0	25.0	22.0	20.0	24.0	1	4		
19.0	23.0	21.0	25.0	22.0	20.0	24.0	1	5		
19.0	23.0	21.0	25.0	22.0	20.0	24.0	1	6		
19.0	23.0	21.0	25.0	22.0	20.0	24.0	1	7		
19.0	21.0	23.0	25.0	22.0	20.0	24.0	2	3	3	21.0
19.0	21.0	23.0	25.0	22.0	20.0	24.0	2	4	3	21.0
19.0	21.0	23.0	25.0	22.0	20.0	24.0	2	5	3	21.0
19.0	20.0	23.0	25.0	22.0	21.0	24.0	2	6	6	20.0
19.0	20.0	23.0	25.0	22.0	21.0	24.0	2	7	6	20.0
19.0	20.0	23.0	25.0	22.0	21.0	24.0	3	4	6	20.0
19.0	20.0	22.0	25.0	23.0	21.0	24.0	3	5	5	22.0
19.0	20.0	21.0	25.0	23.0	22.0	24.0	3	6	6	21.0
19.0	20.0	21.0	25.0	23.0	22.0	24.0	3	7	6	21.0
19.0	20.0	21.0	23.0	25.0	22.0	24.0	4	5	5	23.0
19.0	20.0	21.0	22.0	25.0	23.0	24.0	4	6	6	22.0
19.0	20.0	21.0	22.0	25.0	23.0	24.0	4	7	6	22.0
19.0	20.0	21.0	22.0	23.0	25.0	24.0	5	6	6	23.0
19.0	20.0	21.0	22.0	23.0	25.0	24.0	5	7	6	23.0
19.0	20.0	21.0	22.0	23.0	24.0	25.0	6	7	7	24.0

Fluxograma:**Java:**

```

1. class Exemplo65 {
2.
3.     public static void main (String args []) {
4.         final int diasSemana = 7;
5.         float temperatura [] = new float [diasSemana];
6.         temperatura [0] = 19.0f;
7.         temperatura [1] = 23.0f;
8.         temperatura [2] = 21.0f;
9.         temperatura [3] = 25.0f;
10.        temperatura [4] = 22.0f;
11.        temperatura [5] = 20.0f;
    }
}

```

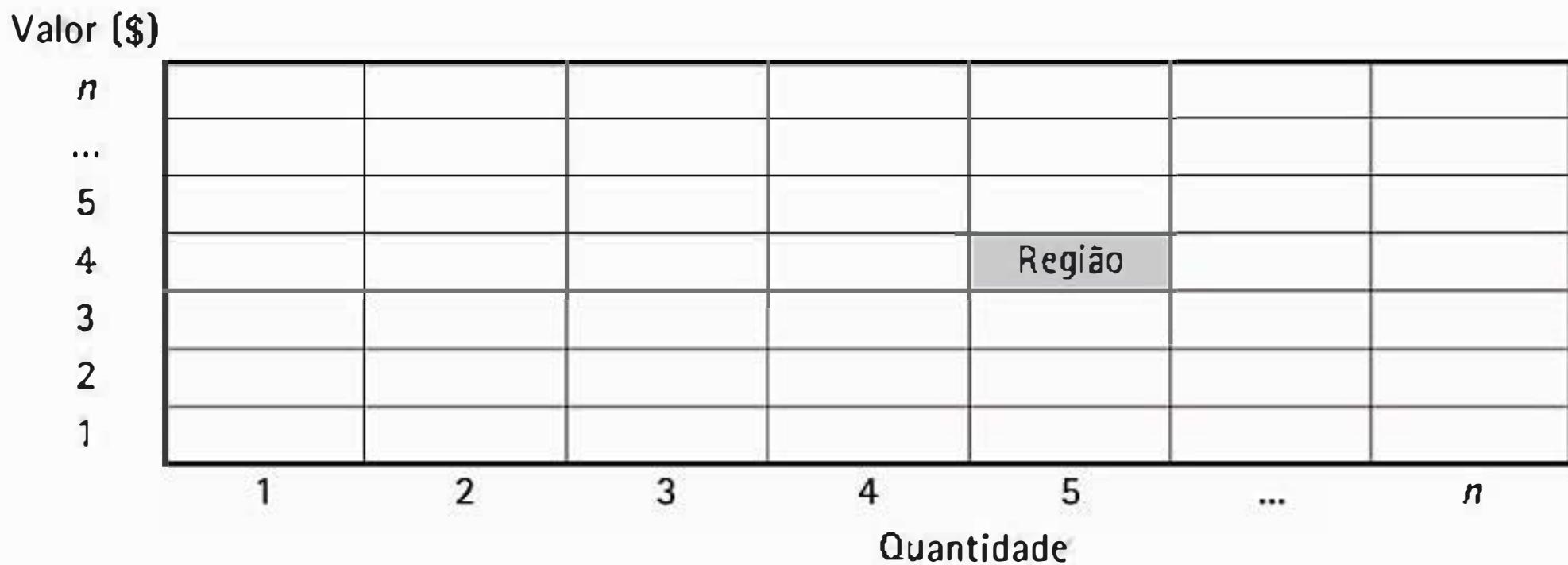
```
12.     temperatura [6] = 24.0f;
13.     float aux;
14.     int min;
15.     for (int i = 0; i < diasSemana - 1 ; i++){
16.         min = i;
17.         for (int j = i + 1; i < diasSemana; i++){
18.             if (temperatura[j] < temperatura[min]){
19.                 min = j;
20.                 aux = temperatura[min];
21.                 temperatura[min] = temperatura [i];
22.                 temperatura[i] = aux;
23.             }
24.         }
25.     }
26.     System.out.println();
27.     System.out.println("Mínima da semana = " + temperatura[0]);
28.     System.out.println("Mínima da semana = " + temperatura[6]);
29.
30. }
31. }
```

6.2 CONCEITO DE MATRIZES

Estruturas indexadas que necessitam de mais que um índice para identificar um de seus elementos são chamadas de **matrizes de dimensão n** , onde n representa o número de índices requeridos. Uma matriz de dimensão 2, portanto, é uma matriz que exige dois índices para identificar um elemento na sua estrutura. A maioria das linguagens não impõe limite sobre a dimensão de uma estrutura indexada, ficando a cargo do programador utilizar tantos índices quantos achar convenientes.

Supondo que se necessite desenhar um gráfico de uma curva no plano e que, portanto, seja necessário guardar as posições dos pontos dessa curva em coordenadas x e y , uma maneira possível de armazenar em memória o total dos pontos dessa curva seria na forma de uma matriz de dimensão 2. Nela, um dado elemento conteria o valor correspondente ao ponto identificado pelo índice de x para a abscissa e y para a ordenada desse elemento.

Por exemplo, se a curva representasse as vendas de um determinado produto numa região, o elemento da linha 4 e coluna 5 conteria a região para as vendas de 5 unidades e para o valor de \$ 4 no período em questão. Essa situação pode ser ilustrada conforme a figura a seguir:



6.2.1 DECLARAÇÃO

A declaração de uma matriz poderia ser feita da seguinte forma:

```
Var
    Vendas : vetor [1..n,1..n] de inteiros
```

Essa declaração é muito semelhante à declaração de vetor porque o vetor é uma matriz de dimensão 1. Delimitadas entre os colchetes, temos duas declarações de tipo associadas aos índices, separadas por uma vírgula. A convenção mais comum é dizermos que o primeiro índice identifica uma linha de uma matriz bidimensional e o segundo, uma coluna. No exemplo anterior, o elemento marcado na matriz seria: `Vendas [4, 5]`.

Se a dimensão n da matriz fosse maior do que dois, então teríamos n declarações de tipos de índices entre os colchetes, separados por vírgulas. A primeira declaração corresponde ao tipo do primeiro índice, a segunda declaração, ao tipo do segundo índice e assim por diante. Uma outra maneira de representar dados agregados pode ser uma estrutura de registro. O registro permite a composição de estruturas do tipo vetor, matriz e dados primitivos, permitindo maior flexibilidade para o programador e um código de leitura mais fácil. As estruturas de registro são apresentadas no Capítulo 7.

A linguagem Java não oferece suporte a arrays (vetores) multidimensionais, a exemplo do que ocorre com outras linguagens de programação. Porém, a mesma funcionalidade pode ser obtida com a declaração de um array de arrays, por exemplo: `int [][]`. Isso será mais detalhado nos exemplos.

6.2.2 OPERAÇÕES

Da mesma forma que se podem fazer operações com os elementos de um vetor, é possível fazê-las com os elementos de uma matriz. É possível acessar individualmente os elementos e, por conseguinte, os valores de cada uma das posições e realizar cálculos matemáticos e comparativos, o que dá uma grande margem de possíveis aplicações computacionais e práticas. As matrizes têm especial aplicação na estatística, pois sua estrutura permite o armazenamento de medições que precisem ser referenciadas a outros valores em duas ou mais dimensões.

Para demonstrar a realização de operações simples em matrizes, será considerado um exemplo genérico. Aplicações específicas serão mostradas em seguida.

EXEMPLO 6.6: Dada uma matriz de 6 linhas e 2 colunas de inteiros, calcular e exibir a média geométrica dos valores de cada uma das linhas. A média geométrica é calculada pela seguinte expressão: $\text{SQRT} (X_1 * X_2)$, que representa a raiz quadrada do resultado da multiplicação dos elementos da coluna 1 (X_1) pelos elementos da coluna 2 (X_2).

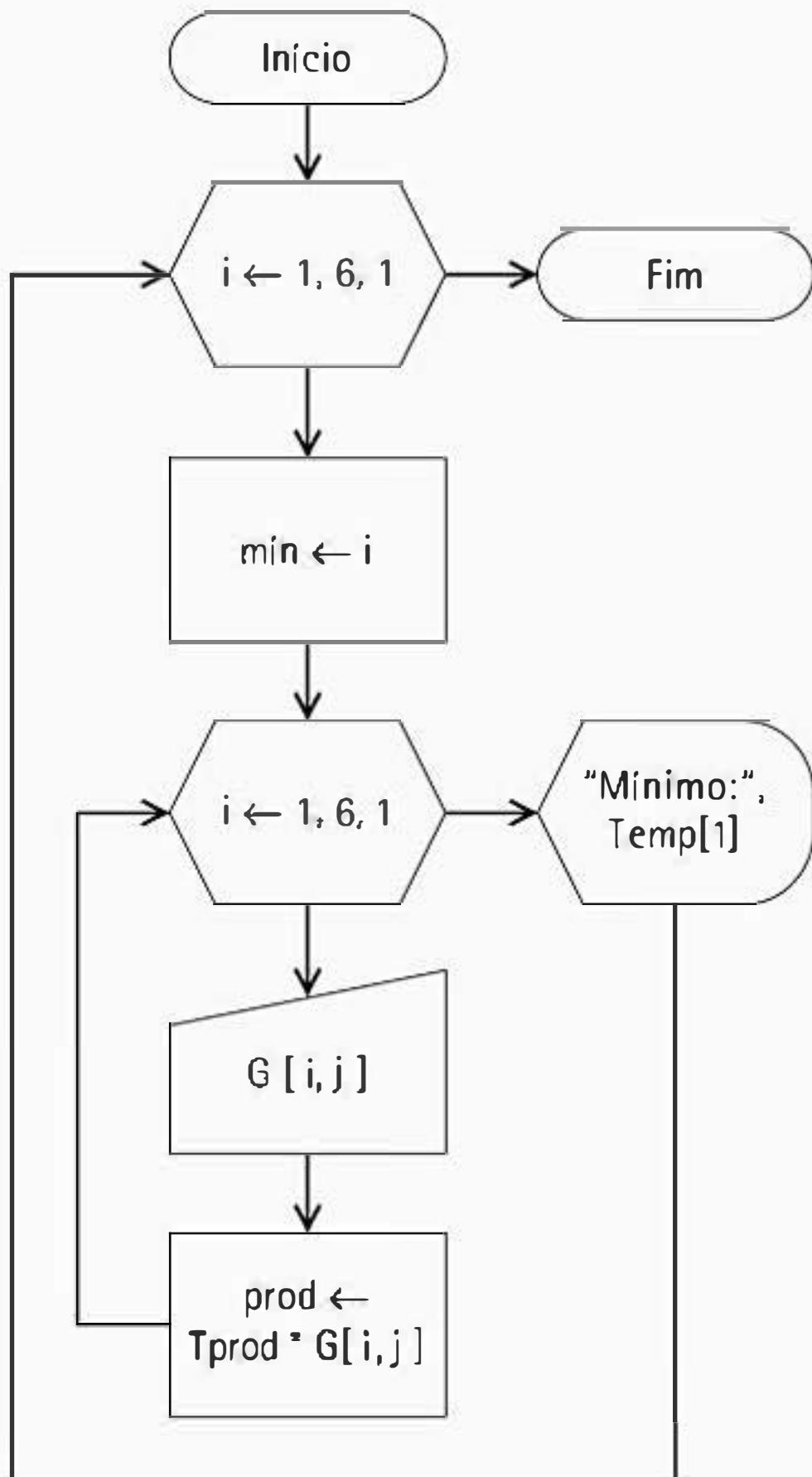
	1	2
1		
2		
3		
4		
5		
6		

Pseudocódigo:

```

1. Algoritmo Exemplo6.6
2. Var
3.     G : vetor [1..6, 1..2] de inteiros
4.     i, j : inteiro
5.     prod : real
6. Início
7.     Para i ← 1 até 6 Faça
8.         prod ← 1
9.         Para j ← 1 até 2 Faça
10.            Ler (G [i, j])
11.            prod ← prod * G [i, j]
12.        Fim-Para
13.        Mostrar ("Linha ", i, " = ", SQRT (prod))
14.    Fim-Para
15. Fim.

```

Fluxograma:**Java:**

```

1. import java.io.*;
2. class Exemplo66{
3.     public static void main (String args[]){
4.         int G [][] = new int [6][2];
5.         double prod;
6.         BufferedReader entrada;
7.         entrada = new BufferedReader(new InputStreamReader
(System.in));
8.         try{
9.             for (int i = 0; i < 6; i++){
10.                 prod = 1;
11.                 for (int j = 0; j < 2; j++){
12.                     System.out.println ("Entre com valores de G-" + i +
", " + j);
13.                     G [i][j] = Integer.parseInt (entrada.readLine());
14.                     prod = prod * G [i][j];
15.                 }
16.                 System.out.println ("Linha-" + i + ":" + Math.sqrt
(prod));
  
```

```

17.         }
18.     } catch (Exception e) {
19.         System.out.println ("Ocorreu um erro durante a
20.         leitura!");
21.     }
22. }
```

Na programação desse exemplo, foi utilizado o pacote `java.lang.Math`, que possui métodos que realizam operações matemáticas mais complexas, como é o caso da raiz quadrada extraída da variável `prod`, utilizando-se a chamada: `Math.sqrt` (linha 16). Não há necessidade de um `import` do pacote, como é feito para o caso do `java.io`, pois ele está automaticamente disponível para a linguagem. Caso isso seja feito, não provocará erro.

EXEMPLO 6.7: Média dos alunos de uma disciplina. Considere uma matriz de 30 linhas e 3 colunas. Cada linha está associada a um aluno de uma determinada disciplina, e as colunas estão associadas às notas das três provas referentes àquele estudante. O procedimento abaixo escreve a média de cada estudante e a média da turma em cada prova.

Pseudocódigo:

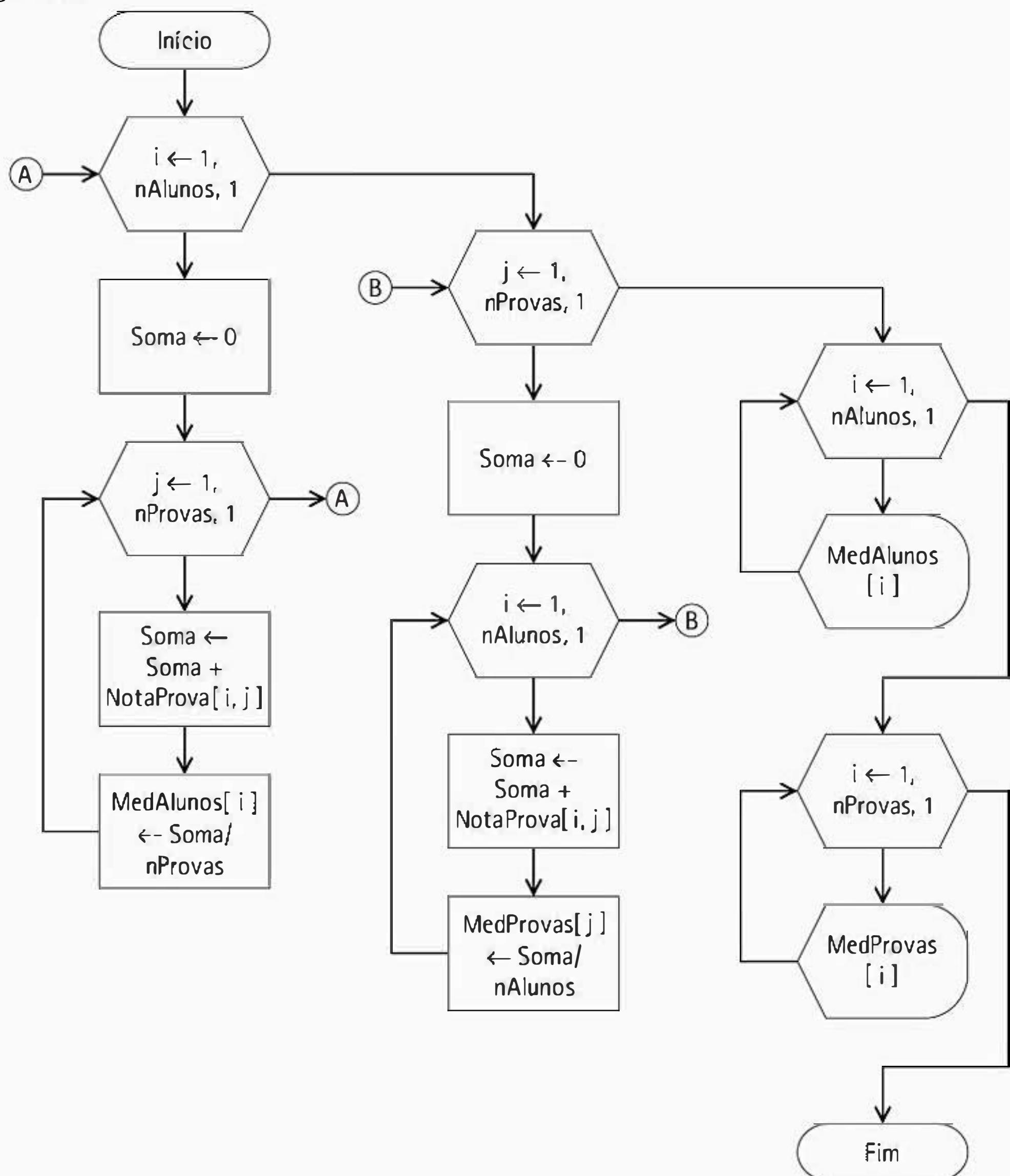
```

1. Algoritmo Exemplo6.7
2. Constante
3.   nProvas = 3 //número de Provas - colunas
4.   nAlunos = 30 //número de Alunos - linhas
5. Var
6.   NotaProva : vetor [1..nAlunos, 1..nProvas] de reais //nota de
   cada prova
7.   MedAlunos : vetor [1..nAlunos] de reais //média dos alunos
8.   MedProvas : vetor [1..nProvas] de reais //média das provas
9.   i, j : inteiro
10.  Soma : real
11. Início
12.  Para i ← 1 até nAlunos Faça
13.    Soma ← 0
14.    Para j ← 1 até nProvas Faça
15.      Soma ← Soma + NotaProva [ i , j ]
16.      MedAlunos [ i ] ← Soma / nProvas
17.    Fim-Para
18.  Fim-Para
19.  Para j ← 1 até nProvas Faça
20.    Soma ← 0
21.    Para i ← 1 até nAlunos Faça
22.      Soma ← Soma + NotaProva [ i , j ]
23.      MedProvas [ j ] ← Soma / nAlunos
24.  Fim-Para
```

```
25.    Fim-para
26.    Para i ← 0 até nAlunos Faça
27.        Mostrar (MedAlunos [i])
28.    Fim-Para
29.    Para i ← 0 até nProvas Faça
30.        Mostrar (MedProvas [i])
31.    Fim-Para
32. Fim.
```

Nos exemplos dados, sabemos quais são as dimensões da matriz e utilizamos todos os elementos da estrutura. Nem sempre, contudo, é possível determinar o tamanho de uma estrutura indexada quando construímos o programa. Em muitos casos, o número exato de elementos necessários só se torna conhecido em tempo de execução. Nesses casos, definimos estruturas que possam acomodar os ‘piores casos’ e utilizamos, em tempo de execução, apenas parte dessas estruturas.

No exemplo das médias de provas e de alunos, poderíamos declarar a matriz com mais colunas para o caso de provas de recuperação ou uma matriz com mais linhas para o caso de novos alunos.

Fluxograma:**Java:**

```

1. import java.io.*;
2. class Exemplo67{
3.     public static void main (String args[]){
4.         final int nProvas = 3;
5.         final int nAlunos = 30;
6.         float NotaProva [][] = new float [nAlunos][nProvas];
7.         float MedAlunos [] = new float [nAlunos];
8.         float MedProvas [] = new float [nProvas];
9.         float Soma;
10.        BufferedReader entrada;

```

```

11.     entrada = new BufferedReader(new
12.         InputStreamReader(System.in));
13.     try{
14.         for (int i = 0; i < nAlunos; i++){
15.             Soma = 0;
16.             for (int j = 0; j < nProvas; j++){
17.                 System.out.println ("Entre com a nota Aluno-" + i +
" Nota-" + j);
18.                 NotaProva [i][j] = Float.parseFloat
(entrada.readLine());
19.                 Soma = Soma + NotaProva [i][j];
20.                 MedAlunos [i] = Soma / nProvas;
21.             }
22.             for (int j = 0; j < nProvas; j++){
23.                 Soma = 0;
24.                 for (int i = 0; i < nAlunos; i++){
25.                     Soma = Soma + NotaProva [i][j];
26.                     MedProvas [j] = Soma / nAlunos;
27.                 }
28.             }
29.             for (int i = 0; i < nAlunos; i++){
30.                 System.out.println ("Media do Aluno-" + i + ":" + 
MedAlunos [i]);
31.             }
32.             for (int i = 0; i < nProvas; i++){
33.                 System.out.println ("Media da Prova-" + i + ":" + 
MedProvas [i]);
34.             }
35.         }catch (Exception e){
36.             System.out.println ("Ocorreu um erro durante a leitura!");
37.         }
38.     }
39. }
```

6.3 EXERCÍCIOS PARA FIXAÇÃO

1. Dadas as temperaturas que foram registradas diariamente durante uma semana, deseja-se determinar em quantos dias dessa semana a temperatura esteve acima da média. A solução para esse problema envolve os seguintes passos:
 - a) obter os valores das temperaturas;
 - b) calcular a média desses valores;
 - c) verificar quantos deles são maiores que a média.

2. Crie vetores para armazenar:
 - a) as letras vogais do alfabeto;
 - b) as alturas de um grupo de dez pessoas;
 - c) os nomes dos meses do ano.
 3. Considere um vetor w cujos nove elementos são do tipo inteiro:

1	2	3	4	5	6	7	8	9
w:								

Supondo que `i` seja uma variável do tipo inteiro e seu valor seja 5, que valores estarão armazenados em `w` após a execução das atribuições a seguir?

- a) $w[1] \leftarrow 17$
 - b) $w[i \text{ div } 2] \leftarrow 9$
 - c) $w[2 * i - 1] \leftarrow 95$
 - d) $w[i-1] := w[9] \text{ div } 2$
 - e) $w[i] := w[2]$
 - f) $w[i+1] := w[i] + w[i-1]$
 - g) $w[w[2]-2] := 78$
 - h) $w[w[i] - 1] \leftarrow w[1] * w[i]$

4. Codifique um algoritmo Histograma que exiba um histograma da variação da temperatura durante a semana. Por exemplo, se as temperaturas forem: 19, 21, 25, 22, 20, 17 e 15°C, de domingo a sábado, respectivamente, o algoritmo deverá exibir:

Suponha ainda que as temperaturas sejam todas positivas e nenhuma seja maior que 80°C. (*Dica:* crie uma rotina que exiba uma linha com uma quantidade de caracteres de tamanho proporcional à temperatura.)

5. Elabore um algoritmo que, considerando um conjunto de acertos obtidos por um grupo de atiradores em um estande, obtenha as discrepâncias e a variância da amostra. Utilize a tabela abaixo como referência. Como exercício, agrupe os valores em uma matriz.

Atirador	Acertos (X_i)	x_i	$(x_i)^2$
1	8		
2	4		
3	6		
4	10		
5	9		
6	7		
7	8		
8	12		
Soma			

As discrepâncias são calculadas por $x_i = X_i - M$, onde X_i é a quantidade de acertos de cada atirador e M a média aritmética dos acertos. A variância S é dada pelo somatório de x_i elevado ao quadrado.

6.4 EXERCÍCIOS COMPLEMENTARES

1. Dados os vetores $A = [15, 44, 23, 1, 0, 18, 17, 37, 35, 54]$ e $B = [32, 115, 48, 55, 51, 0, -48, 85, 15, 99]$, crie algoritmos e programa para gerar uma matriz C a partir da:
 - a) multiplicação dos elementos de A por B ;
 - b) adição dos elementos de A e B ;
 - c) subtração dos elementos de A de B ;
 - d) $A \cup B$.
2. Construa uma matriz $X[10, 3]$ cujos valores deverão ser fornecidos randomicamente e exiba os elementos na ordem inversa à da entrada.
3. Uma empresa deseja saber a média de idade dos freqüentadores de sua praça de alimentação. Para isso, você deve construir uma matriz que seja capaz de armazenar a idade de cem pessoas.
 - a) Seria possível armazenar também os nomes dos participantes da pesquisa? Como resolver esse problema?
 - b) Identifique a maior idade da matriz.
 - c) Calcule a média das idades da matriz.
 - d) Identifique a menor idade da matriz.

PROCEDIMENTOS E FUNÇÕES

- ▶ *Procedimentos*
- ▶ *Funções*
- ▶ *Escopo de variáveis*
- ▶ *Parâmetros*
- ▶ *Passagem de parâmetros*



OBJETIVOS:

Abordar os tópicos: procedimentos, funções e parâmetros. Esses são alguns recursos utilizados para tornar os algoritmos mais eficientes e possibilitar a reutilização de códigos, isto é, o uso de algumas rotinas em vários programas, inclusive com objetivos diferentes.

Conforme estudado no Capítulo 4, a programação estruturada consiste na divisão de um problema em partes, tornando a tarefa mais fácil de ser resolvida, diminuindo assim a extensão dos programas de forma que, se alguma alteração ou acerto forem necessários, isso poderá ser feito mais rapidamente. A cada uma dessas partes é dado o nome de **módulo**. A modularização é uma técnica utilizada para desenvolver algoritmos, denominados módulos, por meio de refinamentos sucessivos.

O refinamento sucessivo nada mais é do que a redução de um problema a um conjunto de tarefas destinadas a solucioná-lo de maneira eficiente. Para cada tarefa, desenvolve-se um algoritmo/programa (módulo) que poderá ser utilizado na solução de outros problemas, pois cada módulo é independente. O gerenciamento das tarefas é feito pelo algoritmo principal ou módulo principal. Esse módulo ‘chama’ ou aciona os outros módulos, que deverão ser escritos por meio de funções ou procedimentos.

NOTA:

A uma das técnicas de refinamentos sucessivos dá-se o nome de **top-down** (de cima para baixo), ou seja, parte-se do problema como um todo, de forma abrangente, e vai se detalhando-o até atingir-se o nível desejado. Outra técnica, não muito utilizada mas que tem a mesma função, é a **bottom-up** (de baixo para cima), isto é, parte-se dos conceitos mais detalhados até se chegar ao objeto desejado, isto é, à solução do problema.

7.1 PROCEDIMENTOS

Um **procedimento** (*procedure*), também conhecido como **sub-rotina**, é um conjunto de instruções que realiza determinada tarefa. Um algoritmo de procedimento é criado da mesma maneira que outro algoritmo qualquer: deve ser identificado, possui variáveis, operações e até funções.

Pseudocódigo:

```
Procedimento nome_do_procedimento {lista de parâmetros}
var
    Declaração dos objetos pertencentes a este procedimento (objetos locais)
inicio
    Instruções do procedimento
Fim Procedimento
```

Nem sempre existe a necessidade do uso de parâmetros. Esse assunto será abordado mais adiante.

Fluxograma:

**NOTA:**

A instrução **retornar** indica que o controle do fluxo de dados deverá retornar ao algoritmo principal.

Java:

Em Java, os módulos, sejam eles procedimentos ou funções, seguem uma sintaxe única, independentemente de qual seja seu objetivo, e são representados pelas classes ou métodos. Para se escrever métodos em Java, utiliza-se a seguinte sintaxe:

```
<modificadores> <tipo_retorno> <nome_Método> (<argumentos>) {  
    <tipo_retorno> <nome_variável_retorno> = <valor_inicial>;  
    <instruções>;  
    return <nome_variável_retorno>;  
}
```

Onde:

- <nome_Método>: é um identificador válido da linguagem. Obedece às mesmas regras que os identificadores de classe, objeto e variável.
- <argumentos>: indica a lista de argumentos que serão passados como parâmetros para o método. A sintaxe dos argumentos é a de declaração de variáveis: *tipo identificador*, e os vários parâmetros são separados por vírgulas.
- <tipo_retorno>: indica o tipo do valor retornado pelo método. Esse tipo pode ser um tipo primitivo, uma classe ou void, no caso de não retornar valor (equivalente a um procedimento).
- return: palavra reservada que indica o valor que será devolvido para o programa, sendo associada com a variável que armazena esse valor. No caso de um método com tipo de retorno void, nada é devolvido, portanto não há retorno e é desnecessário utilizar return.
- <modificadores>: são elementos que caracterizam o método quanto à visibilidade (escopo) e qualidade. Os métodos, como as classes e as variáveis, podem possuir mais de um modificador, não importando sua ordem.

Os modificadores mais utilizados são:

- public: pode ser invocado livremente. Indica um método que é visível para qualquer um que enxergue a classe.
- protected: pode ser utilizado apenas no mesmo pacote e em subclasses.
- private: pode ser invocado apenas na classe.
- final: não pode ser sobreescrito. Equivale à declaração de constante.
- static: não necessita de objeto. Pode ser invocado a partir do nome da classe. Por exemplo: Integer.parseInt(<String>).

Além desses, existem outros modificadores, como por exemplo: abstract, native, transient, volatile e synchronized.

NOTA:

Se não há modificador, o método pode ser chamado apenas no mesmo pacote.

7.1.1 CHAMADA DE PROCEDIMENTOS

A chamada de um procedimento é o momento em que o procedimento é acionado e seu código é executado, isto é, a tarefa associada a ele é realizada pelo algoritmo principal.

Algoritmo principal

Var

Declaração das variáveis utilizadas no algoritmo principal

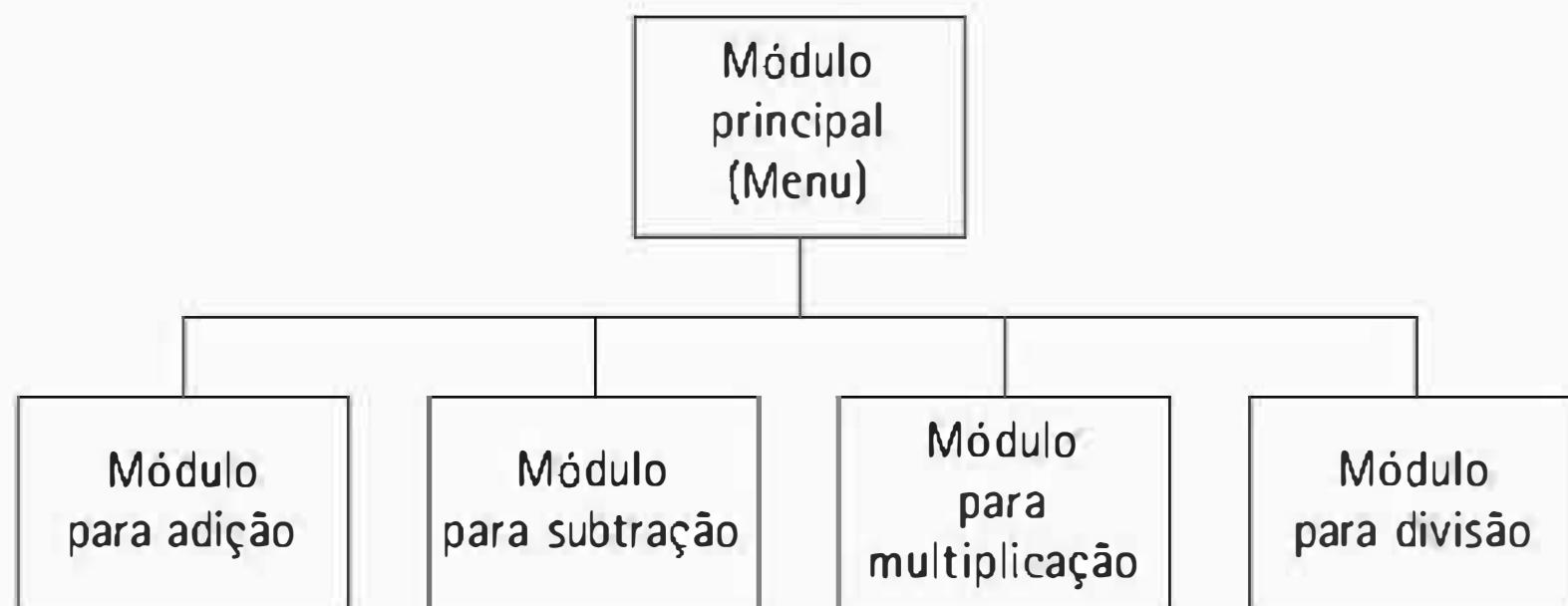
Declaração do procedimento	Procedimento Nome_do_procedimento Var Declaração das variáveis do procedimento Início do procedimento Instruções do procedimento Fim do procedimento
----------------------------	---

Início do algoritmo principal

Instruções do algoritmo principal

Fim do algoritmo principal.

EXEMPLO 7.1: Elaborar um algoritmo que realize a operação aritmética escolhida pelo usuário, a saber: adição, subtração, multiplicação ou divisão, entre dois valores fornecidos por ele.
 Deverá ser criado um menu de opções para o usuário.



Pseudocódigo:

1. Algoritmo Menu
2. Var Opcao: inteiro
3. Inicio
4. Ler (Opcao)
5. Escolha Opcao

```
6.          Caso 1 Então Faça ModAdicao
7.          Caso 2 Então Faça ModSubtr
8.          Caso 3 Então Faça ModMultip
9.          Caso 4 Então Faça ModDiv
10.         Caso contrário Então Mostrar ("Fim de Programa")
11.         Fim_Escolha
12. Fim.
```

No algoritmo Menu, foi criado um menu de opções e, quando determinada opção é escolhida, o procedimento associado a essa opção é acionado e executado. Após a execução do procedimento, o fluxo do programa retoma para o algoritmo principal, nesse caso o algoritmo Menu.

A seguir são apresentados os procedimentos de adição, subtração, multiplicação e divisão que são chamados pelo programa principal.

```
1. Procedimento ModAdicao
2. Var v1, v2, res: real
3. Início
4.   Ler (v1, v2)
5.   res ← v1 + v2
6.   Mostrar (res)
7. Fim.

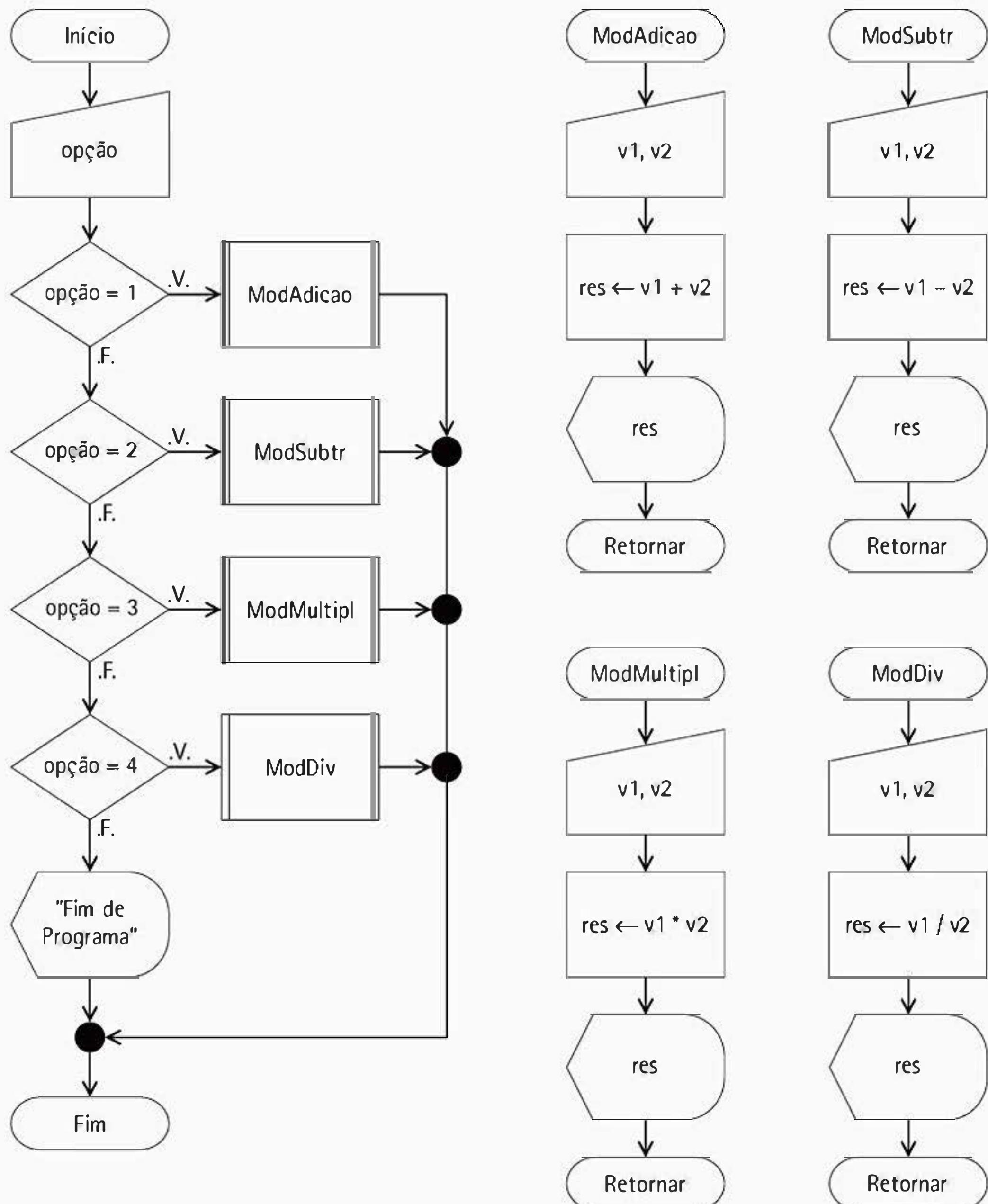
8. Procedimento ModSubtr
9. Var v1, v2, res: real
10. Início
11.   Ler (v1, v2)
12.   res ← v1 - v2
13.   Mostrar (res)
14. Fim.

15. Procedimento ModMultip
16. Var v1, v2, res: real
17. Início
18.   Ler (v1, v2)
19.   res ← v1 * v2
20.   Mostrar (res)
21. Fim.

22. Procedimento ModDiv
23. Var v1, v2, res: real
24. Início
25.   Ler (v1, v2)
26.   res ← v1 / v2
27.   Mostrar (res)
28. Fim.
```

Observe que, nos procedimentos anteriores, as variáveis `v1`, `v2` e `res` são declaradas em todos os algoritmos, pois estão declaradas como variáveis locais. Esse problema pode ser resolvido declarando-as como variáveis globais no módulo principal. Esse assunto será detalhado mais adiante na Seção “Escopo de variáveis”. Cabe lembrar que esses procedimentos podem ser utilizados em outros algoritmos e programas.

Fluxograma:



Java:

```
1. import java.io.*;
2.
3. class Menu {
4.
5.     public static void main (String args []) {
6.         BufferedReader entrada;
7.         Entrada = new BufferedReader(
8.             new InputStreamReader(System.in));
9.         Try {
10.             System.out.println("1 : Adicao");
11.             System.out.println("2 : Subtracao");
12.             System.out.println("3 : Multiplicacao");
13.             System.out.println("4 : Divisao");
14.             System.out.println("Qual a Opcão Desejada?");
15.             int opcao = Integer.parseInt(entrada.readLine());
16.             switch (opcao){
17.                 case 1 : modAdicao(); break;
18.                 case 2 : modSubtracao(); break;
19.                 case 3 : modMultiplicacao(); break;
20.                 case 4 : modDivisao();break;
21.                 default : System.out.println("Fim do Programa");
22.             }
23.         } catch (Exception erro) {
24.             System.out.println("Ocorreu um erro de leitura!");
25.         }
26.     }
27.
28.     static void modAdicao() {
29.         BufferedReader entraSoma;
30.         entraSoma = new BufferedReader(
31.             new InputStreamReader(System.in));
32.         Try {
33.             System.out.println("Qual o primeiro numero?");
34.             float numero1 = Float.parseFloat(entraSoma.readLine());
35.             System.out.println("Qual o segundo numero?");
36.             float numero2 = Float.parseFloat(entraSoma.readLine());
37.             float resultado = numero1 + numero2;
38.             System.out.println("Soma = " + resultado);
39.         } catch(Exception erro){
40.             System.out.println("Ocorreu um erro de leitura!");
41.         }
42.     }
43.
44.     static void modSubtracao () {
45.         BufferedReader entraSub;
46.         entraSub = new BufferedReader(
47.             new InputStreamReader(System.in));
```

```

48.     Try {
49.         System.out.println("Qual o primeiro numero?");
50.         float numero1 = Float.parseFloat(entradaSub.readLine());
51.         System.out.println("Qual o segundo numero?");
52.         float numero2 = Float.parseFloat(entradaSub.readLine());
53.         float resultado = numero1 - numero2;
54.         System.out.println("Subtracao = " + resultado);
55.     } catch(Exception erro){
56.         System.out.println("Ocorreu um erro de leitura!");
57.     }
58. }

59.

60. static void modMultiplicacao () {
61.     BufferedReader entraMult;
62.     entraMult = new BufferedReader(
63.             new InputStreamReader(System.in));
64.     Try {
65.         System.out.println("Qual o primeiro numero?");
66.         float numero1 = Float.parseFloat(entraMult.readLine());
67.         System.out.println("Qual o segundo numero?");
68.         float numero2 = Float.parseFloat(entraMult.readLine());
69.         float resultado = numero1 * numero2;
70.         System.out.println("Multiplicacao = " + resultado);
71.     } catch(Exception erro){
72.         System.out.println("Ocorreu um erro de leitura!");
73.     }
74. }

75.

76. static void modDivisao () {
77.     BufferedReader entraDiv;
78.     entraDiv = new BufferedReader(
79.             new InputStreamReader(System.in));
80.     Try {
81.         System.out.println("Qual o primeiro numero?");
82.         float numero1 = Float.parseFloat(entraDiv.readLine());
83.         System.out.println("Qual o segundo numero?");
84.         float numero2 = Float.parseFloat(entraDiv.readLine());
85.         float resultado = numero1 / numero2;
86.         System.out.println("Divisao = " + resultado);
87.     } catch(Exception erro){
88.         System.out.println("Ocorreu um erro de leitura!");
89.     }
90. }
91. }

```

NOTA:

A ordem em que os métodos são escritos é indiferente para a execução do programa, da mesma forma que esta independe da ordem em que eles são chamados. O importante é a ordem em que a lógica do programa necessita executar tais métodos.

7.2 FUNÇÕES

As funções são criadas e chamadas da mesma maneira que os procedimentos. A diferença entre eles é que as funções podem ser utilizadas em expressões, como se fossem variáveis, pois as funções retornam valores que são associados ao seu nome e para esses valores se faz necessária a declaração do tipo de dado a ser retornado.

```

Função nome_da_função (lista de parâmetros): tipo_de dado da
função
var
    Declaração dos objetos pertencentes a esta função (objetos
    locais)
    inicio
        Instruções da função
    Retornar (variável)

```

NOTA: Tanto os procedimentos como as funções são ‘mini-algoritmos’ que possuem variáveis e até mesmo outros procedimentos e funções.

Fluxograma:



Como já foi explicado na Seção “Procedimentos”, na linguagem de programação Java, tanto os procedimentos quanto as funções são tratados como métodos.

EXEMPLO 7.2: Ler um número fornecido pelo usuário e calcular o fatorial.

1. Função Fatorial(n : inteiro) : inteiro
2. Var
3. i, fat: inteiro
4. Início
5. Ler(n)
6. Se (n = 0) Então
7. Fatorial ← 1

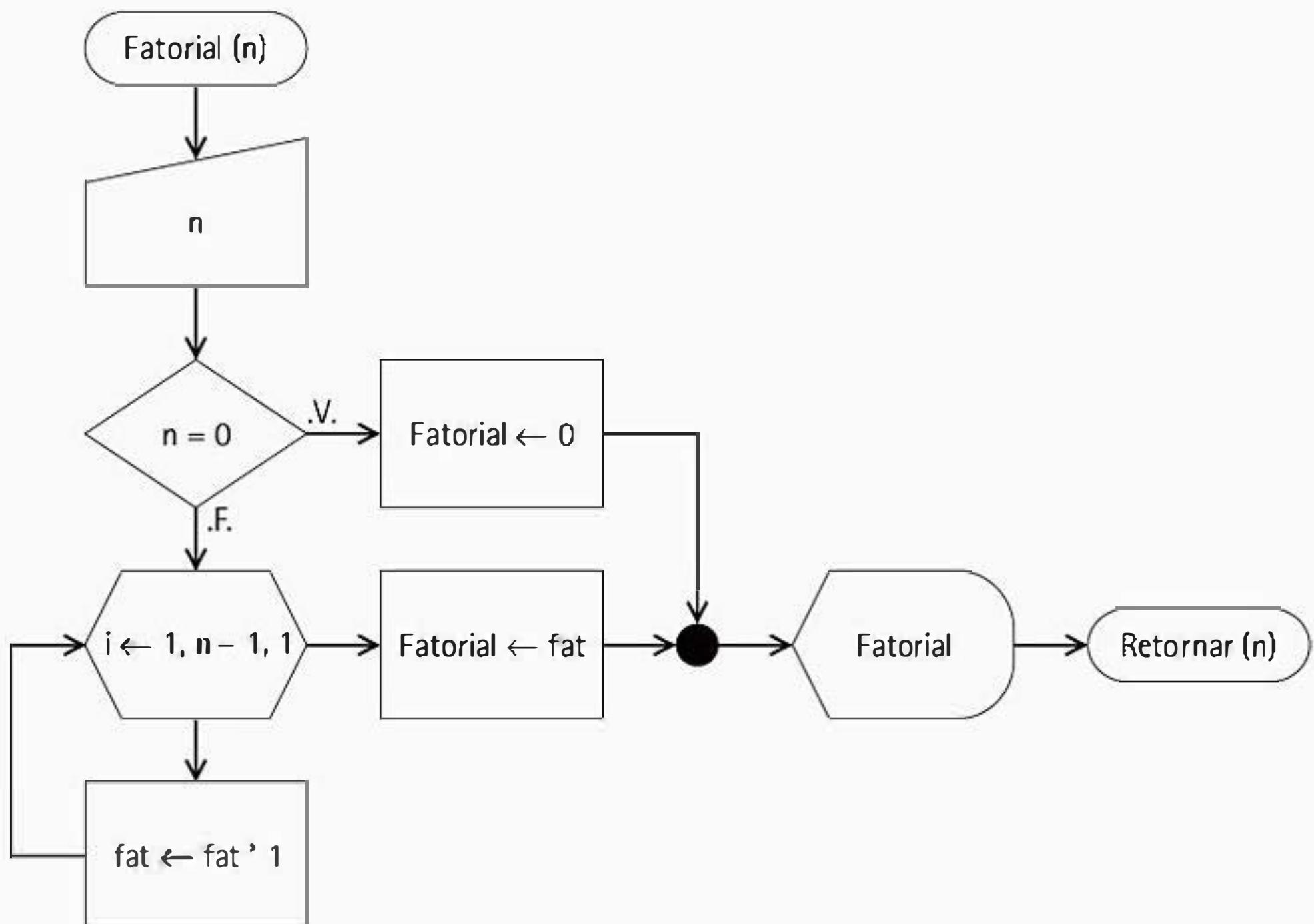
```

8.    Senão
9.        Para i de 1 até n - 1 passo 1 Faça
10.            fat ← fat * i
11.        Fim-Para
12.        Fatorial ← fat
13.    Fim-Se
14.    Retornar (Fatorial)
15. Fim.

```

NOTA:

A declaração do tipo de dado da função e do tipo de dado da variável de parâmetro da função se faz necessária, pois o parâmetro passado (saiba mais a seguir) pode ser de um tipo e o retorno do resultado de outro tipo.

Fluxograma:**Java:**

```

1. import java.io.*;
2. class Exemplo72{
3.     public static void main (String args[]){
4.         BufferedReader entrada;
5.         entrada = new BufferedReader (new InputStreamReader
(System.in));
6.         try{
7.             System.out.println ("Qual Número?");
8.             int numero = Integer.parseInt (entrada.readLine());
9.             int fat = fatorial (numero);

```

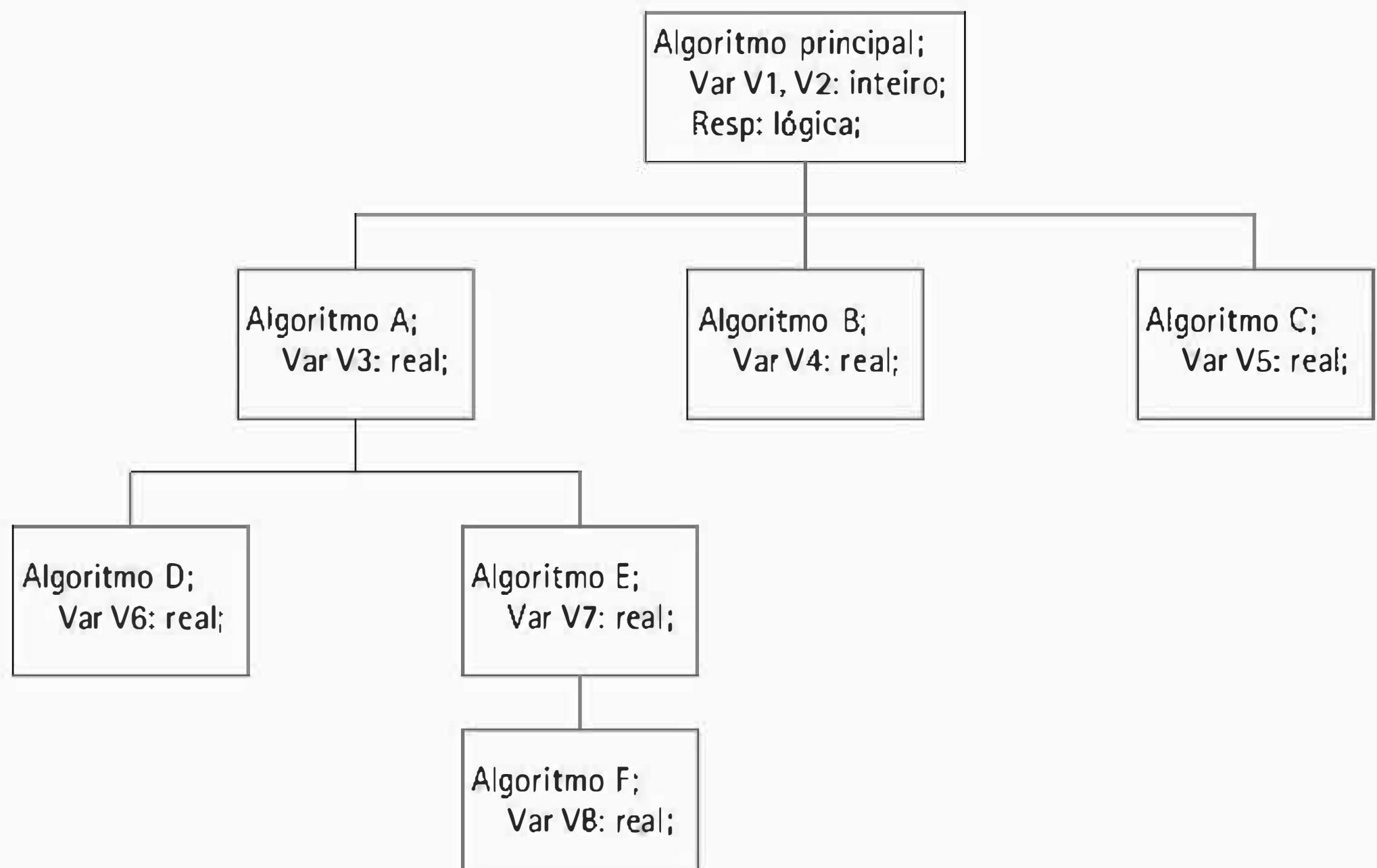
```

10.         System.out.println ("fatorial = " + fat);
11.     }catch (Exception erro){
12.         System.out.println ("Ocorreu um erro de leitura!");
13.     }
14. }
15. static int factorial (int num){
16.     int fat = 1;
17.     for (int i = 1; i <= num; i++)
18.         fat = fat * i;
19.     return fat;
20. }
21. }
```

7.3 ESCOPO DE VARIÁVEIS

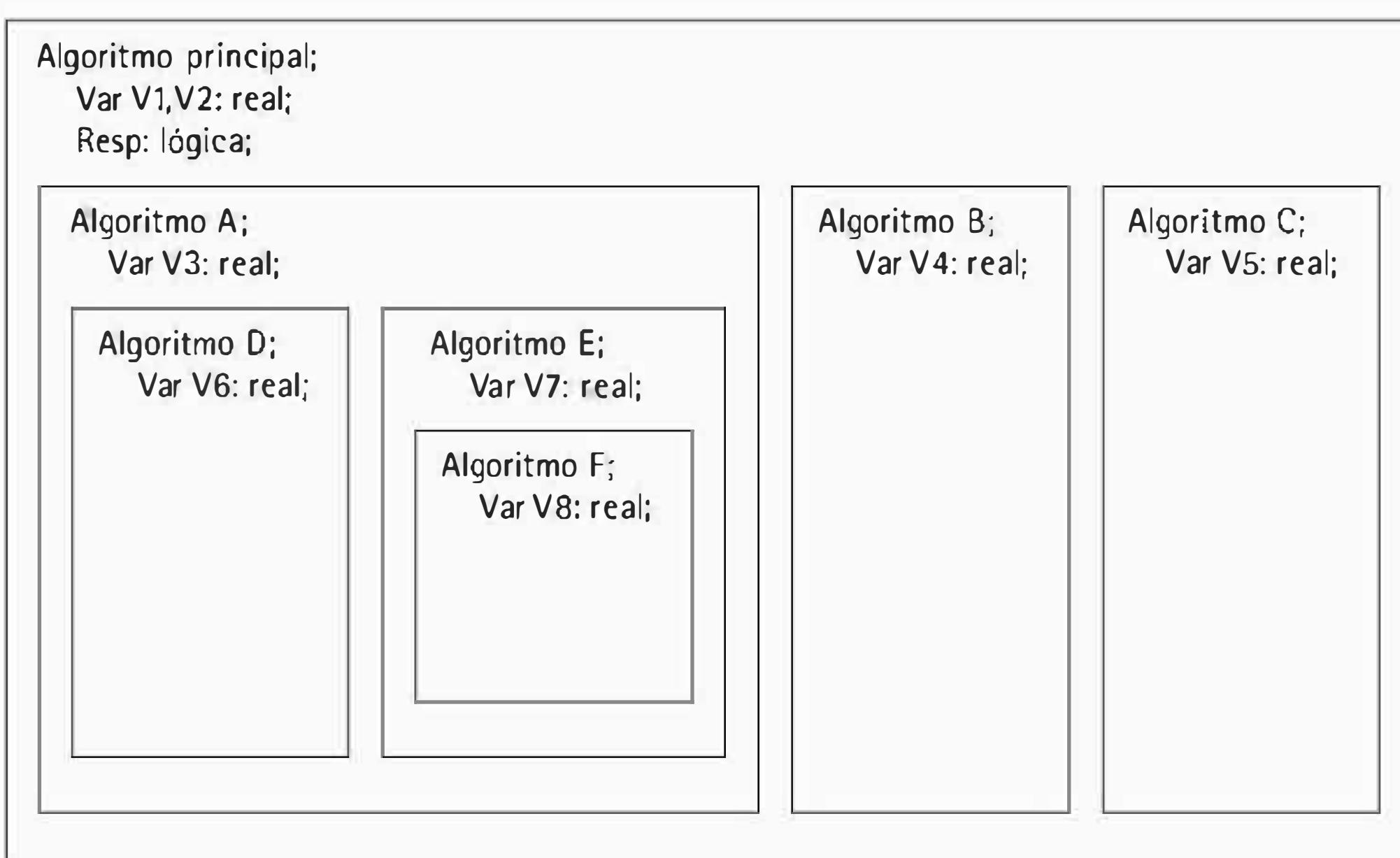
Até agora, todos os exemplos apresentados utilizaram variáveis locais, isto é, variáveis que podem ser utilizadas somente no escopo do algoritmo/programa no qual foram declaradas. No entanto, isso pode ocasionar redundância na declaração de variáveis que se fazem necessárias em vários módulos, como no caso do Exemplo 7.1, em que as variáveis `v1`, `v2` e `res` foram declaradas em todos os módulos. Para resolver esse problema, existe a possibilidade de serem declaradas variáveis globais.

As variáveis globais são declaradas no algoritmo principal e podem ser utilizadas por todos os algoritmos hierarquicamente inferiores. Já as variáveis locais podem ser utilizadas pelo algoritmo em que foram declaradas e nos algoritmos hierarquicamente inferiores, conforme mostra o organograma apresentado a seguir:



Supondo que esse organograma represente a hierarquia da resolução de um problema em módulos, tem-se que:

- As variáveis V1 e V2 foram declaradas no módulo principal e podem ser utilizadas por todos os módulos dos algoritmos.
- A variável V3 foi declarada no algoritmo A e pode ser utilizada pelos algoritmos D, E e F, que são hierarquicamente inferiores a ele.
- As variáveis V4, V5, V6 e V8 podem ser utilizadas somente pelos algoritmos B, C, D e F, respectivamente, pois não possuem algoritmos hierarquicamente inferiores.
- A variável V7 pode ser utilizada pelos algoritmos E e F.


NOTA:

A definição adequada das variáveis pode economizar memória e tornar os programas mais eficientes.

7.4 PARÂMETROS

Parâmetros são variáveis ou valores que podem ser transferidos do algoritmo principal para um módulo que está sendo chamado. Eles funcionam como comunicadores entre os módulos. Existem dois tipos de parâmetros: os formais e os reais.

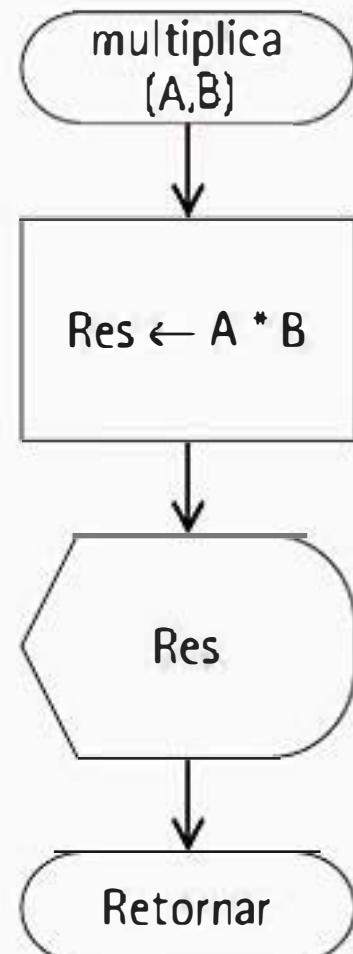
7.4.1 PARÂMETROS FORMAIS

São declarados nos módulos e tratados como as variáveis. O algoritmo que chama a função ou o procedimento informa os valores que substituirão esses parâmetros. No exemplo a seguir, as variáveis A e B são **parâmetros formais**.

EXEMPLO 7.3: Calcular a multiplicação entre dois números.

1. Algoritmo Exemplo_7.3
2. Procedimento multiplica(A, B: reais)
3. Var Res: real
4. Início
5. Res ← (A * B)
6. Mostrar (Res)
7. Fim.

Fluxograma:



Java:

```

1. static float multiplicar (float A, float B) {
2.     float res;
3.     res = A * B;
4.     return res;
5. }
  
```

7.4.2 PARÂMETROS REAIS

Os **parâmetros reais** são os valores que substituem os parâmetros formais. O algoritmo que chama a função ou o procedimento informa esses valores ou variáveis. No Exemplo 7.4 os parâmetros formais A e B do Procedimento multiplicar serão substituídos pelos valores fornecidos para as variáveis num1 e num2 do algoritmo principal.

EXEMPLO 7.4: Calcular a multiplicação entre dois números.

1. Algoritmo Exemplo_7.4
2. Procedimento multiplicar (A, B: real)

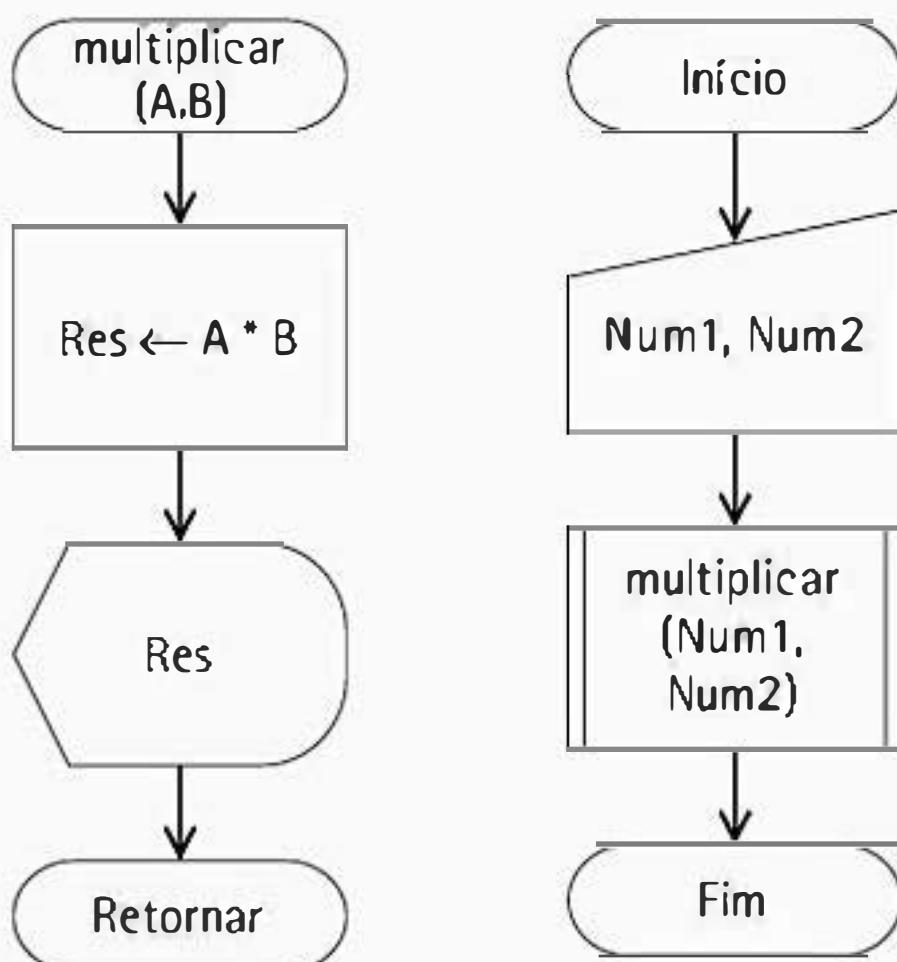
```

3.      Var Res: real
4.      Início
5.          Res ← (A * B)
6.          Mostrar (Res)
7.      Fim.
8.
9.      Var Num1, Num2: real
10.     Início
11.        Ler (Num1, Num2)
12.        multiplicar (Num1, Num2)
13.    Fim.

```

As variáveis Num1 e Num2 são os parâmetros reais, isto é, são as variáveis que receberão valores que serão utilizados na execução da função ou procedimento, no caso do exemplo anterior, para a realização da multiplicação, portanto, quando o procedimento multiplicar (linha 12) é chamado, informa os valores, neste caso num1 e num2, que substituirão os parâmetros formais, neste caso A e B.

Fluxograma:



LEMBRETE:

Se o procedimento multiplicar fosse implementado como uma função, seria necessário informar o tipo de dado esperado como retorno na declaração da função e deveria ser utilizada a palavra retorno (variável), no algoritmo, ou return (variável) no programa Java.

Java:

```

1. import java.io.*;
2.
3. class Multiplicar {
4.

```

```
5.     public static void main (String args []) {
6.         BufferedReader entrada;
7.         entrada = new BufferedReader(
8.             new InputStreamReader(System.in));
9.         try {
10.             System.out.println("Qual Número 1?");
11.             int num1 = Integer.parseInt(entrada.readLine());
12.             System.out.println("Qual Número 2?");
13.             int num2 = Integer.parseInt(entrada.readLine());
14.             multiplicar (num1, num2);
15.         } catch (Exception erro) {
16.             System.out.println("Ocorreu um erro de leitura!");
17.         }
18.     }
19.
20.     static void multiplicar (float A, float B) {
21.         float res;
22.         res = A * B;
23.         System.out.println(" " + res);
24.     }
25. }
```

7.4.3 PASSAGEM DE PARÂMETROS

A **passagem de parâmetros** ocorre por meio da correspondência argumento/parâmetro, em que os argumentos são valores constantes ou variáveis informados no módulo chamador, ou principal — isto é, passagem de parâmetros é a substituição dos parâmetros formais pelos parâmetros reais. Os argumentos devem ser fornecidos na mesma ordem dos parâmetros.

Os parâmetros podem ser passados por valor ou por referência.

7.4.4 PASSAGEM DE PARÂMETROS POR VALOR

Na **passagem de parâmetros por valor**, o valor do parâmetro real é copiado para o parâmetro formal do módulo, preservando, assim, o valor original do parâmetro.

No Exemplo 7.4, foi utilizado esse tipo de parâmetro. Foi solicitada uma entrada para os parâmetros Num1 e Num2 (que são parâmetros reais) e estes foram armazenados em A e B (que são parâmetros formais), os quais foram manipulados, preservando-se assim os valores de Num1 e Num2.

NOTA:

Na linguagem de programação Java, os objetos (String, Date etc.) e os arrays são sempre passados por referência. O Java passa o valor da variável quando o método é chamado e quaisquer alterações feitas na variável tornam-se permanentes.

7.4.5 PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Na **passagem de parâmetros por referência**, toda alteração feita nos parâmetros formais reflete-se nos parâmetros reais; assim, o parâmetro é de entrada e saída. No Exemplo 7.4, as variáveis `Num1` e `Num2` são parâmetros de entrada e a variável `Total` é um parâmetro de saída.

7.5 EXERCÍCIOS PARA FIXAÇÃO

Para os exercícios abaixo, escreva os procedimentos ou funções adequados para cada problema.

1. Escreva um algoritmo para ordenar três números fornecidos pelo usuário. Utilize a passagem de parâmetros formais.
2. Escreva um algoritmo para:
 - a) preencher uma matriz A ;
 - b) ordenar os elementos da matriz A ;
 - c) gerar uma matriz somente com os números pares da matriz A ;
 - d) gerar uma matriz somente com os números múltiplos de 5;
 - e) criar um menu para acessar os itens anteriores.
3. Elabore um algoritmo que seja capaz de fazer a conversão de um valor em reais para o correspondente em dólares, libras, francos e ienes e vice-versa. O programa deverá conter um menu com as opções.
4. Construa um algoritmo que verifique se um dado número é divisível por outro. Ambos devem ser fornecidos pelo usuário.
5. Faça um algoritmo que possibilite o arredondamento de um número real para um número inteiro seguindo os padrões científicos.
6. Elabore um algoritmo que escreva por extenso um número inteiro com até dez dígitos fornecido pelo usuário.
7. Construa um algoritmo que verifique, sem utilizar a função `mod`, se um número é divisível por outro.

7.6 EXERCÍCIOS COMPLEMENTARES

1. Escreva um algoritmo que calcule o máximo divisor comum de dois números fornecidos pelo usuário.
2. Necessita-se calcular alguns dados correspondentes aos animais de uma fazenda. Os animais pertencem a espécies diferentes, a saber: bovinos, ovinos e caprinos. Construa um algoritmo para a média de peso de cada espécie para os animais do sexo feminino e do sexo masculino, a partir de dados fornecidos pelo usuário.

3. Dada uma data abreviada, escreva-a por extenso.
4. Faça a leitura de mil números pelo processo de sorteio automático. Os números devem estar entre 0 e 100. Verifique:
 - a) Qual foi o número sorteado mais vezes.
 - b) Qual foi o número sorteado menos vezes.
 - c) Qual foi o maior número.
 - d) Qual foi o menor número.
5. Construa um algoritmo que calcule a somatória dos N primeiros números de um conjunto. O valor de N deverá ser fornecido pelo usuário.
6. Escreva um algoritmo que calcule o número de horas de determinado período estabelecido por duas datas.

BUSCA E ORDENAÇÃO

- ▶ *Ordenação*
- ▶ *Busca*



OBJETIVOS:

Abordar algumas técnicas para construção de algoritmos para ordenação e busca de dados.

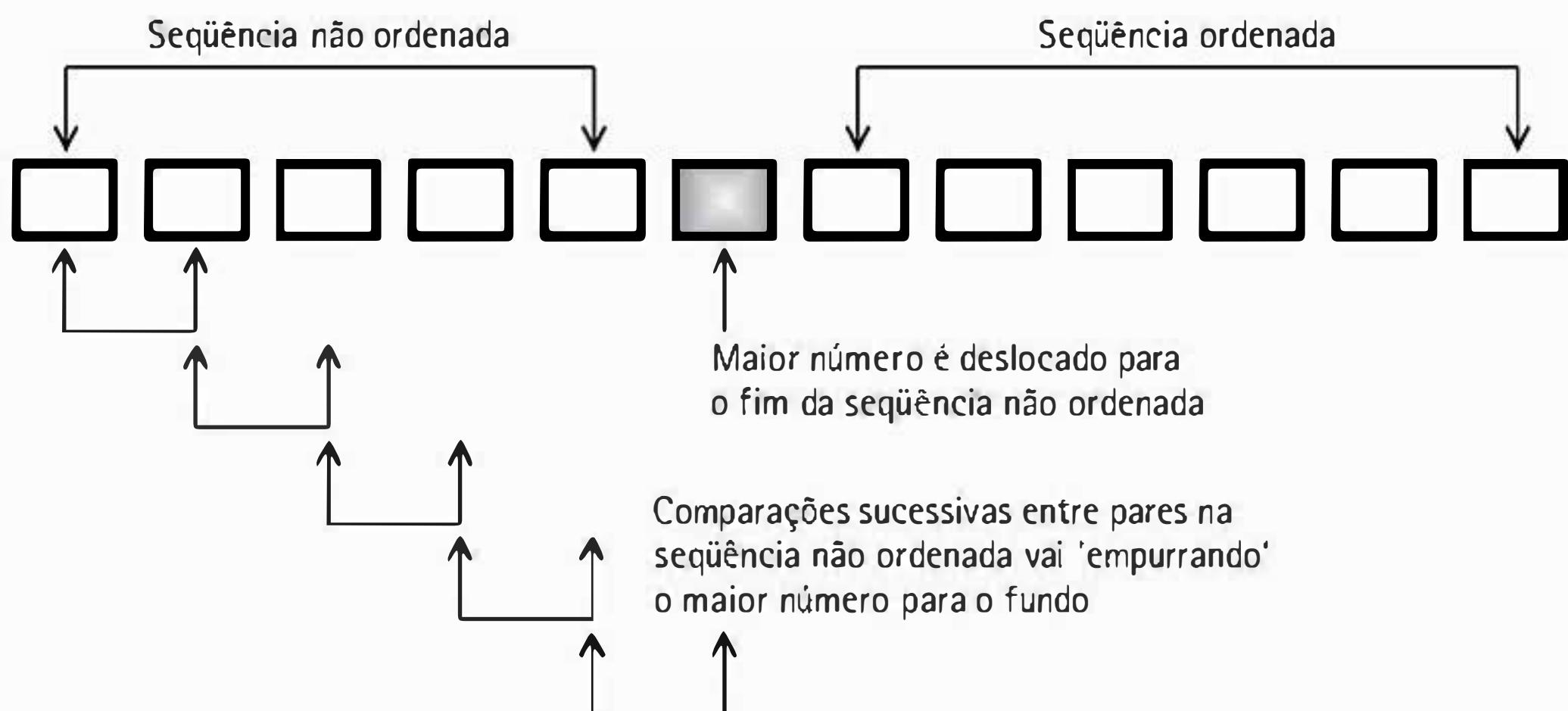
Nos capítulos anteriores, foi mostrado como trabalhar com dados fornecidos das mais diversas formas pelos usuários. Normalmente, o usuário que vai inserir os dados não está ou não pode estar preocupado com a ordem de entrada dos dados no momento de sua inserção na relação, de modo que é comum encontrarmos elementos ordenados de maneira aleatória em nossos sistemas.

Muitas vezes, necessitamos que esses dados apresentem uma ordem para que possamos realizar ações como verificar se determinado cliente pagou uma conta, se uma pessoa está em uma lista de convidados para uma festa e assim por diante. Devido a essas necessidades, foram desenvolvidos vários algoritmos de ordenação que consistem, basicamente, em realizar comparações sucessivas e trocar os elementos de posição.

Neste capítulo, mostraremos alguns dos métodos de ordenação básicos.

8.1 ORDENAÇÃO POR TROCAS — MÉTODO DA BOLHA

O método de **ordenação por trocas** é considerado o mais simples de todos. Consiste em comparar pares consecutivos de valores e permutá-los caso estejam fora de ordem. O algoritmo determina uma seqüência de comparações sistemáticas que varrem a seqüência de dados como um todo, fazendo com que o maior valor (ou menor, de acordo com a ordem desejada) acabe no final da seqüência e uma nova série de comparações sistemáticas se inicia.



|FIGURA 8.1| Ordenação pelo Método da Bolha

Em cada passagem, um elemento é deslocado para sua posição final, isto é, um elemento é ordenado. Assim, uma seqüência com N elementos terá, após a primeira passagem, um elemento ordenado e $N - 1$ elementos por ordenar. Na segunda passagem, a seqüência terá dois elementos ordenados e $N - 2$ elementos por ordenar e assim sucessivamente.

A idéia desse tipo de ordenação é análoga à idéia de jogar pedras na água. Enquanto as pedras (elementos mais pesados) vão para o fundo, as bolhas de ar (elementos mais leves) vão para a superfície. Daí o nome do método ser conhecido como **Método da Bolha**.

EXEMPLO 8.1: Algoritmo de ordenação por trocas — Método da Bolha

1. Algoritmo Exemplo8.1
2. Var
3. numeros : vetor de inteiros
4. aux, i, j : inteiro
5. Início
6. Para i de <início> até <fim - 1> Faça
7. Para j de <início> até <fim - 1 - i> Faça
8. Se (numeros[j] > numeros[j + 1])

```

9.          aux = numeros[j]
10.         numeros [j] = numeros [j + 1]
11.         numeros[j + 1] = aux
12.     Fim-Se
13.   Fim-Para
14. Fim-Para
15. Fim.

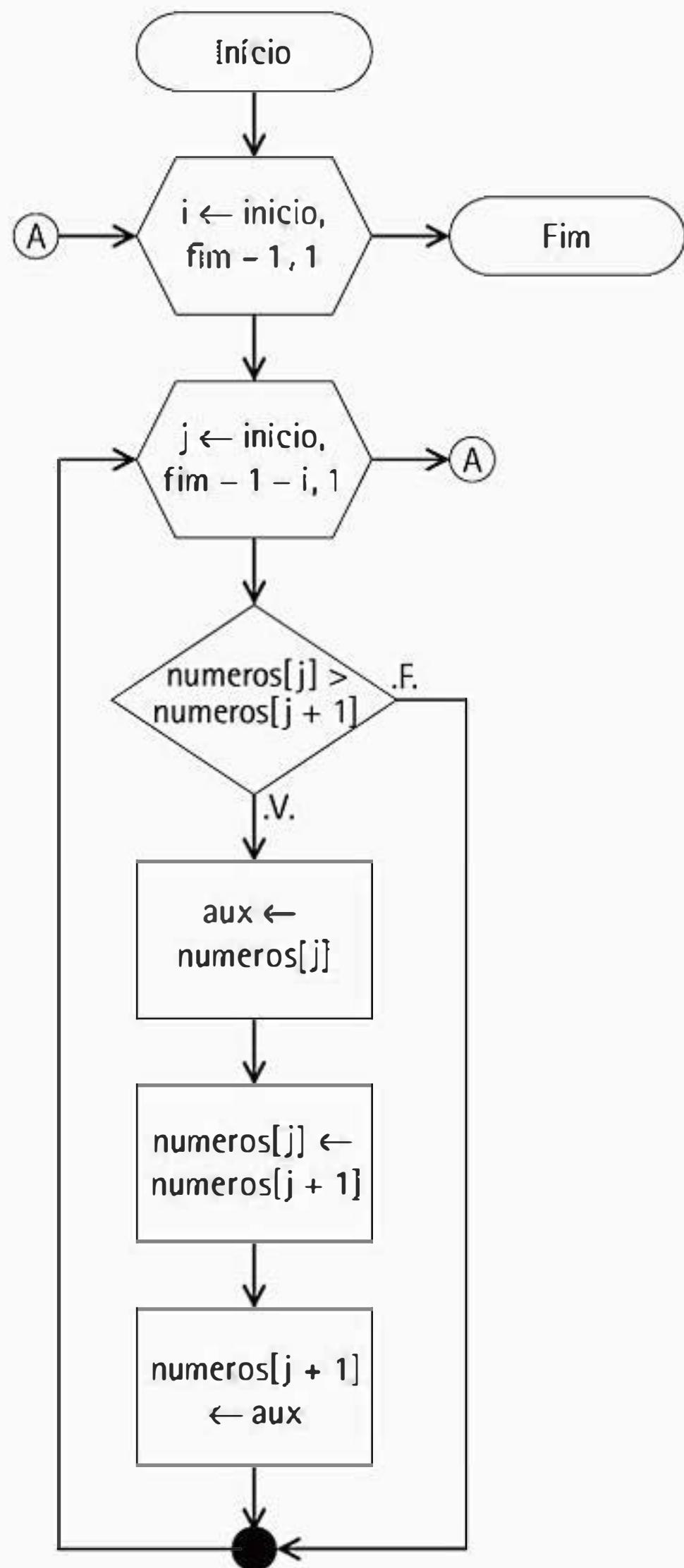
```

Na linha 6, a variável *i* terá o valor inicial definido pela expressão <início> e o valor final definido pela expressão <fim - 1>. O algoritmo faz uma série de passagens do início até o fim da seqüência que se deseja ordenar, também na linha 6. Como os dois últimos números serão ordenados simultaneamente (quando o penúltimo estiver ordenado, o último também estará), a estrutura de repetição irá do índice do primeiro elemento até o índice do penúltimo.

Em cada passagem, serão feitos um conjunto de comparações sucessivas (linhas 8 a 12) e a troca dos valores que estiverem fora de ordem. Nesse caso, o número de comparações é determinado por uma estrutura de repetição (linha 7) que vai desde o primeiro até o último elemento não ordenado, que é dado pelo índice do último elemento da seqüência a ser ordenada (*fim - 1*) menos a quantidade de elementos que já foram ordenados (*i*).

Observe que a variável auxiliar utilizada para a permuta dos elementos deve ser do mesmo tipo que os elementos que estão sendo permutados.

Considerando que cada passagem ordena um elemento por meio de comparações sucessivas entre os elementos não ordenados, uma seqüência de *n* elementos terá $n - 1$ passagens (a última passagem ordena dois números ao mesmo tempo) e cada passagem terá $n - i$ comparações, em que *i* é o número da passagem.

Fluxograma:

Vamos considerar, para o Exemplo 8.1, que o vetor contém os seguintes elementos: [9, 1, 3, 2, 7, 5, 4]. Então, substituindo as variáveis <início> e <fim – 1>, teríamos a seguinte representação no pseudocódigo (linhas 6 e 7) e no fluxograma:

Para i de 0 até 5 faça
 Para j de 0 até S – i faça

Na Tabela 8.1 são apresentados, passo a passo, os valores ordenados a cada execução do laço, considerando o vetor exemplo [9, 1, 3, 2, 7, 5, 4].

Essa tabela apresenta os valores de um vetor carregado inicialmente com os valores representados pela coluna 0. Cada coluna subsequente representa os passos do algoritmo necessários para ordenar totalmente o vetor. As linhas nomeadas com *i* e *j* representam os valores que essas variáveis assumiriam no decorrer do processo.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
i	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	3	3	3	4	4	5
j	0	0	1	2	3	4	5	0	1	2	3	4	0	1	2	3	0	1	2	0	1	0
0	9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	9	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	3	3	9	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	2	2	2	9	7	7	7	7	7	7	5	5	5	5	5	4	4	4	4	4	4	4
4	7	7	7	7	9	5	5	5	5	5	7	4	4	4	4	5	5	5	5	5	5	5
5	5	5	5	5	5	9	4	4	4	4	7	7	7	7	7	7	7	7	7	7	7	7
6	4	4	4	4	4	4	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

TABELA 8.1 | Ordenação de um vetor

É interessante observar que, nesse caso, o algoritmo já concluiu a ordenação no 15º passo, o que não ocorreria com outra ordem de entrada dos valores (coluna 0). Esse método de tabular os dados pode ser utilizado para outros exemplos, facilitando a compreensão e a verificação da eficácia do algoritmo. Não utilizaremos essa representação para todos os casos, cabendo ao leitor aplicá-la quando julgar necessário.

EXEMPLO 8.2: O programa a seguir implementa uma solução para ordenar o vetor de números inteiros [9, 1, 3, 2, 7, 5, 4] utilizando o método de ordenação por trocas.

Java:

```

1. class Exemplo82{
2.     public static void main(String args[]){
3.         int num [] = new int [7];
4.         num [0] = 9;
5.         num [1] = 1;
6.         num [2] = 3;
7.         num [3] = 2;
8.         num [4] = 7;
9.         num [5] = 5;
10.        num [6] = 4;
11.        bolha (num);
12.        for (int i = 0; i < num.length; i++)
13.            System.out.println (num [i]);
14.    }
15.    public static void bolha (int numeros[]){
16.        final int n = numeros.length;
17.        int aux;
18.        for (int i = 0; i < n-1; i++){
19.            for (int j = 0; j < n-1-i; j++){
20.                if (numeros[j] > numeros[j+1]){
21.                    aux = numeros[j];
22.                    numeros[j] = numeros[j+1];
23.                    numeros[j+1] = aux;
24.                }
25.            }
26.        }
27.    }
}

```

```

24.         }
25.     }
26.   }
27. }
28. }
```

Esse método descreve exatamente o mesmo que o algoritmo Exemplo 8.2. A identificação do método (linha 15) utiliza o modificador `public`, pois é interessante que esse método tenha máxima visibilidade, embora isso não seja obrigatório. O `void` é utilizado pois é equivalente a um procedimento, isto é, ordena um conjunto de números mas, após seu processamento, nada é acrescentado à memória do sistema. O modificador `static` é usado para torná-lo um método sem dependências com objetos.

Os elementos que se deseja ordenar são passados como um vetor de parâmetros do tipo desejado, cujo tamanho é definido pela propriedade `length` dos vetores em Java, usada na linha 16. O modificador `final` (equivalente a uma declaração de constante) apenas indica que o tamanho do vetor não será alterado durante a ordenação.

CUIDADO!

Ao se fazerem ordenações ou mesmo pesquisas em Java utilizando strings, deve-se dedicar especial atenção. String é uma classe definida na linguagem cujos elementos possuem características especiais. As comparações entre objetos são feitas de maneira diferente das comparações de tipos primitivos (como números inteiros).

Para comparar a ordem de duas strings, existe um método chamado `compareTo()`, definido na classe `String`. Esse método lê o código UNICODE equivalente das duas strings e subtrai um valor do outro. O resultado é um valor numérico do tipo inteiro tal que, se for igual a zero, as duas strings serão iguais; se for menor que zero, elas estarão ordenadas de forma crescente; e, se for maior do que zero, elas estarão em ordem decrescente ou fora de ordem crescente (vide Tabela 8.2).

Para ilustrar a explicação sobre ordenação pelo Método da Bolha para strings, é apresentado a seguir o código em Java cuja única diferença para o Exemplo 8.2 está na condicional da linha 20. Justamente por isso, não existe a necessidade de repetirmos a representação algorítmica em pseudocódigo e fluxograma.

Sintaxe da comparação	Resultado	Significado
<code>String1.compareTo(String2)</code>	<code>= 0</code>	são iguais
<code>String1.compareTo(String2)</code>	<code>> 0</code>	String1 é maior que String2 (fora de ordem crescente)
<code>String1.compareTo(String2)</code>	<code>< 0</code>	String1 é menor que String2 (em ordem crescente)

| TABELA 8.2 | Comparações com strings

EXEMPLO 8.3: Programa para ordenação de caracteres ou strings de caracteres.

```

1. class Exemplo83{
2.     public static void main (String args []){
3.         String letras [] = new String [7];
4.         letras [0] = "AG";
5.         letras [1] = "AA";
6.         letras [2] = "AE";
7.         letras [3] = "AF";
8.         letras [4] = "AC";
9.         letras [5] = "AB";
10.        letras [6] = "B";
11.        bolha (letras);
12.        for (int i = 0; i < letras.length; i++)
13.            System.out.println (letras [i]);
14.    }
15.    public static void bolha (String palavras[]){
16.        final int n = palavras.length;
17.        String aux;
18.        for (int i = 0; i < n - 1; i++){
19.            for (int j = 0; j < n - 1 - i; j++){
20.                if (palavras[j].compareTo(palavras[j + 1])>0){
21.                    aux = palavras[j];
22.                    palavras[j] = palavras[j + 1];
23.                    palavras[j + 1] = aux;
24.                }
25.            }
26.        }
27.    }
28. }
```

Os dois métodos podem ser escritos na mesma classe. Embora os tipos dos parâmetros sejam diferentes, devido à qualidade de sobrecarga dos nomes dos métodos, apenas o método correto responderá à chamada do programa principal.

Outra observação importante deve ser feita. Na ordenação de palavras, o programador deve lembrar que maiúsculas e minúsculas são diferentes, estando as maiúsculas listadas na frente das minúsculas nas tabelas de caracteres.

Para evitar problemas, é recomendável utilizar algum mecanismo de segurança, por exemplo, converter todos os caracteres indicados pelo usuário para maiúsculas ou minúsculas. Em Java, utilizam-se os métodos `toUpperCase()` e `toLowerCase()` para fazer as conversões para maiúsculas e para minúsculas, respectivamente.

O exemplo a seguir mostra um programa completo que permite ordenar um conjunto de elementos do tipo número inteiro ou `String` dados pelo usuário. Nesse exemplo, o usuário deve saber com antecedência a quantidade de elementos que deseja ordenar.

EXEMPLO 8.4: Programa completo para ordenação.

```
1. class Ordenacao {
2.     public static void bolha (int numeros[]){
3.         final int n = numeros.length;
4.         int aux;
5.         for (int i = 0; i < n - 1; i++){
6.             for (int j = 0; j < n - 1 - i; j++){
7.                 if (numeros[j] > numeros[j + 1]){
8.                     aux = numeros[j];
9.                     numeros[j] = numeros[j + 1];
10.                    numeros[j+1] = aux;
11.                }
12.            }
13.        }
14.    }
15.    public static void bolha (String palavras[]){
16.        final int n = palavras.length;
17.        String aux;
18.        for (int i = 0; i < n - 1; i++){
19.            for (int j = 0; j < n - 1 - i; j++){
20.                if (palavras[j].compareTo(palavras[j + 1])>0){
21.                    aux = palavras[j];
22.                    palavras[j] = palavras[j + 1];
23.                    palavras[j + 1] = aux;
24.                }
25.            }
26.        }
27.    }
28. }

1. import java.io.*;
2.
3. class Exemplo84{
4.     public static void main(String args[]){
5.         BufferedReader entrada;
6.         entrada = new BufferedReader (new InputStreamReader
7.             (System.in));
8.         try{
9.             int op;
10.            boolean ok;
11.            do{
12.                System.out.println("- Ordenar -");
13.                System.out.println("O que deseja ordenar?");
14.                System.out.println("1 - Numeros Inteiros");
15.                System.out.println("2 - Palavras");
16.                System.out.print("Opção -");
```

```
16.         int opcao = Integer.parseInt( entrada.readLine() );
17.         switch (opcao) {
18.             case 1 :
19.                 System.out.print("Quantos Numeros? ");
20.                 qte = Integer.parseInt( entrada.readLine() );
21.                 int numeros [] = new int [qte];
22.                 for (int i = 0; i < qte; i++){
23.                     System.out.print("Numero[" + (i + 1) + "] = ");
24.                     numeros[i] = Integer.parseInt
25.                         (entrada.readLine());
26.                 }
27.                 Ordenacao.bolha(numeros);
28.                 System.out.println();
29.                 System.out.println("Ordenados: ");
30.                 System.out.println();
31.                 for (int i = 0; i < qte; i++)
32.                     System.out.println(numeros[i]);
33.                 ok = true;
34.                 break;
35.             case 2 :
36.                 System.out.print("Quantas Palavras? ");
37.                 qte = Integer.parseInt( entrada.readLine() );
38.                 String palavras [] = new String [qte];
39.                 for (int i = 0; i < qte; i++){
40.                     System.out.print("Palavra[" + (i + 1) + "] = ");
41.                     palavras[i] = entrada.readLine().toUpperCase();
42.                 }
43.                 Ordenacao.bolha(palavras);
44.                 System.out.println();
45.                 System.out.println("Ordenados: ");
46.                 System.out.println();
47.                 for (int i = 0; i < qte; i++)
48.                     System.out.println(palavras[i]);
49.                 ok = true;
50.                 break;
51.             default :
52.                 System.out.println("Opção Inválida!");
53.                 System.out.println("Tente Novamente.");
54.                 System.out.println();
55.                 ok = false;
56.             }
57.         } while (!ok);
58.     }catch (Exception e){
59.         System.out.println("Ocorreu um erro durante a leitura!");
60.     }
61. }
```

Nesse exemplo, a classe Ordenacao possui métodos que permitem ordenar números inteiros e strings, mas o programador pode ampliá-la para permitir a ordenação dos demais tipos primitivos, como números reais (`float`) e caracteres (`char`).

As chamadas dos métodos ocorrem nas linhas 26 e 42. Essas chamadas são realizadas indicando-se o nome da classe em que o método está definido (`Ordenacao`) e o nome do método com os parâmetros necessários. Na linha 40, é feita uma conversão da entrada de dados do usuário de forma que todas as palavras (ou frases) sejam convertidas para letras maiúsculas. Isso foi feito para evitar o problema já mencionado de diferença de ordenação para maiúsculas e minúsculas. O programa apresenta um menu de escolha para o usuário e uma estrutura adequada para uma entrada de dados conveniente.

8.2 BUSCA

Evidentemente, possuir os dados não ajuda o programador ou o usuário se eles não souberem onde os dados estão. Imagine, por exemplo, uma festa de casamento com cem convidados na qual não se sabe quem eles são ou se determinada pessoa foi ou não convidada. Imagine, nas eleições, que você queira votar naquele único político honesto que conhece, mas não sabe qual é seu número!

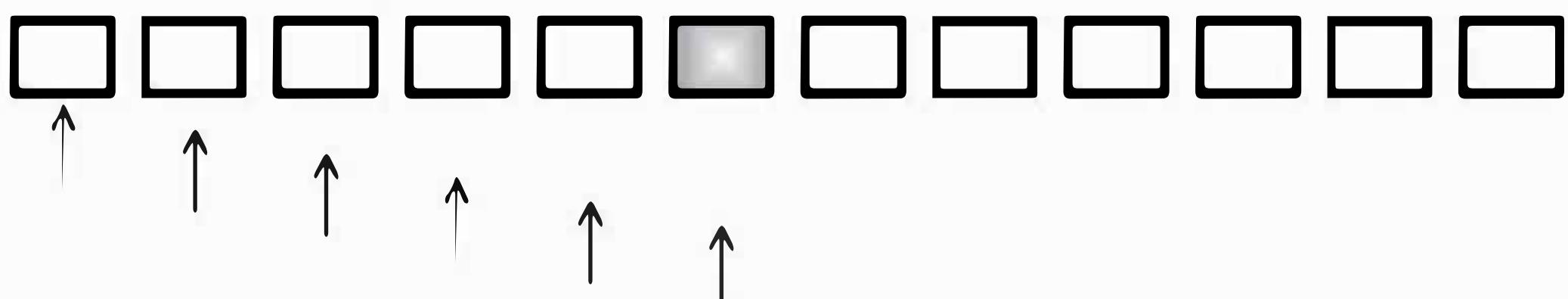
Os algoritmos de busca são alguns dos mais utilizados no mundo da informática, sendo usados em bancos de dados, internet, jogos, entre outros. Aqui serão apresentados alguns exemplos de algoritmos de **busca linear** e **busca binária**.

A escolha do método a ser utilizado para busca depende muito da quantidade de dados envolvidos, do volume de operações de inclusão e exclusão a serem realizadas, entre outros fatores que devem ser considerados quando do desenvolvimento da aplicação.

8.2.1 BUSCA LINEAR (OU SEQÜENCIAL)

A maneira mais óbvia de fazer uma busca é, com certeza, comparar o elemento que se está procurando com todos os elementos guardados, um a um, isto é, o mais simples é procurar o elemento seqüencialmente até que seja encontrado.

O algoritmo que realiza essa busca é realmente muito simples e consiste em uma estrutura de repetição que varre toda a seqüência de elementos, realizando uma condicional que compara o elemento desejado com os elementos existentes na seqüência.



|FIGURA 8.2| Busca linear

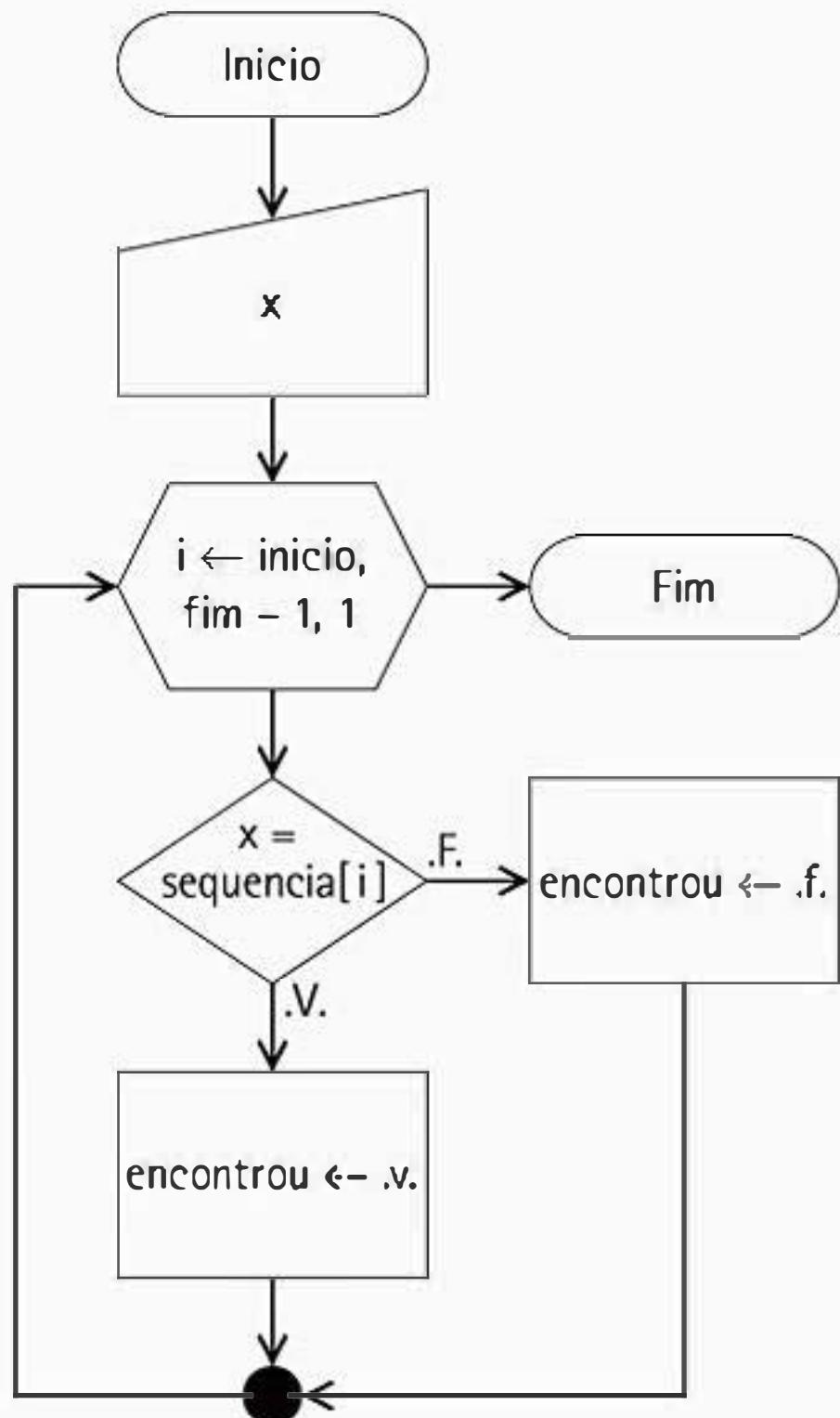
Quando o elemento é encontrado, retorna-se o valor verdadeiro, o que indica o sucesso da busca e encerra a estrutura de repetição. É claro que a maneira de encerrar essa estrutura dependerá da linguagem de programação a ser utilizada.

LEMBRE-SE: A execução da rotina de busca termina quando a condição de busca é satisfeita, ou então quando todo o conjunto é percorrido e o elemento não foi encontrado.

EXEMPLO 8.5: Algoritmo de busca linear (seqüencial).

```
1. Algoritmo Exemplo8.5
2. Var
3.     sequencia : vetor de elementos
4.     x         : elemento a ser procurado
5.     encontrou : logica
6.     i         : inteiro
7. Início
8.     Ler (x)
9.     Para i de <início> até <fim - 1> Faça
10.        Se (x = sequencia[i])
11.            encontrou ← .v.
12.        <fim da busca>
13.        Senão
14.            encontrou ← .f.
15.        Fim-Se
16.    Fim-Para
17. Fim.
```

No algoritmo Exemplo8.5, para a variável `sequencia`, devem-se declarar o tamanho do vetor e o tipo de dados que esse vetor receberá, conforme visto no Capítulo 6, assim como para a variável `x` deve-se declarar o tipo de dado da informação a ser procurada.

Fluxograma:

Uma variação bastante comum dessa e das demais estruturas de busca retorna o índice do elemento procurado na seqüência. Para isso, basta substituir a variável lógica por uma variável de tipo inteiro que receberá o índice do elemento, quando este for encontrado, e poderá receber um valor não associado a nenhum índice, por exemplo, -1 . A variação escolhida sempre será uma opção do programador que refletirá suas necessidades.

O código em Java é tão simples quanto o algoritmo. Uma classe com esse método aplicado a números inteiros é mostrada no programa a seguir.

Java:

```

1. class Busca{
2.
3.     public static boolean linear (int x, int numeros []){
4.         final int n = numeros.length;
5.         for (int i = 0; i < n; i++)
6.             if (x == numeros[i])
7.                 return true;
8.         return false;
9.     }
10. }
```

Na linha 6 do exemplo, é feita a comparação efetiva entre o elemento procurado e os elementos da seqüência. Caso o elemento procurado seja encontrado, o programa retorna **verdadeiro** (linha 7). Em Java, quando a palavra reservada **return** é encontrada, o método é encerrado e as comparações param.

CUIDADO!

Novamente, deve-se atentar para o fato de as comparações entre strings (objetos) serem diferentes das comparações entre tipos primitivos. Um cuidado extra é requerido, pois a utilização do operador de igualdade (==) não provocará erro no programa. Esse operador pode ser utilizado para comparar os objetos em si, por exemplo, para verificar qual objeto foi responsável pela chamada de um método em um programa com interface gráfica. Nesse caso, deseja-se conhecer o objeto (botão, caixa de edição...), e não seu conteúdo (cor do botão, tamanho da caixa de edição...).

Além do método já citado, a classe **String** possui alguns métodos de comparação direta entre duas strings. O método **equals (Object obj)** é derivado da superclasse **Object** e pode ser utilizado para comparar o conteúdo completo de dois objetos quaisquer (inclusive **String**). Esse método é bastante utilizado, mas, para nossos fins de busca, é mais conveniente utilizar o método **equalsIgnoreCase (String texto)**, que faz a comparação apenas entre objetos do tipo **String**, mas elimina o problema de comparações entre maiúsculas e minúsculas.

EXEMPLO 8.6: Programa em Java para busca linear de strings.

```

1. public static boolean linear (String x, String palavras[]){
2.     final int n = palavras.length;
3.     for (int i = 0; i < n; i++)
4.         if (x.equalsIgnoreCase(palavras[i]))
5.             return true;
6.     return false;
7. }
```

Observe o método de busca **linear** descrito para **String**. As alterações estão apenas no identificador do método, que utiliza parâmetros corretos para uma busca em uma seqüência de strings, e na linha 4, onde a comparação foi modificada para adequar-se aos objetos do tipo **String**. Esse e outros métodos de busca linear para os demais tipos primitivos podem ser adicionados à classe descrita no Exemplo 8.5, o que constitui um bom exercício de fixação.

O Exemplo 8.7 retrata um programa completo que faz a busca de um número qualquer determinado pelo usuário em uma seqüência de números inteiros também fornecidos pelo usuário.

EXEMPLO 8.7: Programa completo de busca.

```
1. class Busca{  
2.     public static boolean linear (int x, int dados []){  
3.         final int n = dados.length;  
4.         for (int i = 0; i < n; i++)  
5.             if (x == dados[i])  
6.                 return true;  
7.         return false;  
8.     }  
9.     public static boolean linear (String x, String dados []){  
10.        final int n = dados.length;  
11.        for (int i = 0; i < n; i++)  
12.            if (x.equalsIgnoreCase (dados [i]))  
13.                return true;  
14.        return false;  
15.    }  
16. }
```

A classe Busca incorpora os dois métodos de busca: para números inteiros e para strings. Como já dissemos anteriormente, os métodos estão sobrecarregados, sendo diferenciados pelo tipo dos dados passados como parâmetros.

```
1. import java.io.*;  
2.  
3. class Exemplo87{  
4.     public static void main (String args[]){  
5.         BufferedReader entrada;  
6.         entrada = new BufferedReader (new InputStreamReader  
(System.in));  
7.         try{  
8.             int qte;  
9.             boolean ok, achou;  
10.            do{  
11.                System.out.println ("– Entrada de Dados –");  
12.                System.out.println ("Que tipo deseja usar?");  
13.                System.out.println ("1 - Numeros Inteiros");  
14.                System.out.println ("2 - Palavras");  
15.                System.out.print ("Opcão –");  
16.                int opcao = Integer.parseInt (entrada.readLine());  
17.                switch (opcao){  
18.                    case 1 :  
19.                        System.out.print ("Quantos Numeros?");  
20.                        qte = Integer.parseInt (entrada.readLine());  
21.                        int numeros [] = new int [qte];  
22.                        for (int i = 0; i < qte; i++){  
23.                            System.out.print ("Número[" + (i+1) + "]= ");
```

```
24.         numeros [i] = Integer.parseInt
  (entrada.readLine());
25.     }
26.     System.out.println();
27.     System.out.print("Qual Numero Deseja Pesquisar?");
28.     int x = Integer.parseInt (entrada.readLine());
29.     achou = Busca.linear(x, numeros);
30.     if (achou)
31.         System.out.println("Numero Presente na
  Relacao");
32.     else
33.         System.out.println("Numero Nao Presente na
  Relacao");
34.     ok = true;
35.     break;
36. case 2 :
37.     System.out.print("Quantas Palavras?");
38.     qte = Integer.parseInt( entrada.readLine() );
39.     String palavras [] = new String [qte];
40.     for (int i = 0; i < qte; i++){
41.         System.out.print("Palavra [" + (i+1) + "] = ");
42.         palavras [i] = entrada.readLine().toUpperCase();
43.     }
44.     System.out.println ();
45.     System.out.print ("Qual Palavra Deseja
  Pesquisar?");
46.     String c = entrada.readLine();
47.     achou = Busca.linear(c, palavras);
48.     if (achou)
49.         System.out.println("Palavra Presente na
  Relacao");
50.     else
51.         System.out.println("Palavra Nao Presente na
  Relacao");
52.     ok = true;
53.     break;
54. default :
55.     System.out.println("Opcao Invalida!");
56.     System.out.println("Tente Novamente.");
57.     System.out.println();
58.     ok = false;
59.   }
60. } while (!ok);
61. }catch (Exception e){
62.     System.out.println("Ocorreu um erro durante a leitura!");
63.   }
64. }
65. }
```

Nesse exemplo, que é semelhante ao Exemplo 8.4, pede-se para o usuário identificar o tipo de dado que deseja manipular. Dependendo da opção escolhida, aciona-se o conjunto de instruções adequado.

8.2.2 BUSCA BINÁRIA (OU BUSCA LOGARÍTMICA)

O método de busca linear é o mais adequado quando não se tem nenhuma informação a respeito da estrutura em que a busca será realizada. Por outro lado, se o elemento procurado estiver entre os últimos ou não estiver no conjunto, esse método poderá ser demasiadamente lento, pois ocorrerão comparações com todos os elementos de um conjunto que pode ser muito grande — imagine a relação dos clientes de um banco!

Quando temos uma seqüência ordenada de elementos, existem outros métodos de busca que são muito mais adequados, pois permitem realizar uma busca por meio de algoritmos mais eficientes, que podem utilizar um número menor de comparações.

Considere uma seqüência ordenada de elementos de qualquer tipo. Em vez de se comparar o elemento procurado ao primeiro elemento da seqüência, pode-se compará-lo a um elemento do meio da seqüência. Se o elemento comparado é o desejado, a busca termina; senão, pode-se verificar se o elemento desejado é maior ou menor que o elemento encontrado. Como todos os elementos estão ordenados, essa verificação elimina a metade da seqüência onde o elemento não pode estar. Por exemplo, se o elemento procurado for maior que o elemento encontrado, a metade inferior da lista poderá ser descartada para a próxima comparação.

A segunda comparação será feita com o elemento do meio da seqüência que restou e o procedimento anterior se repetirá. Dessa forma, cada vez que o elemento não for encontrado, o conjunto a ser pesquisado será reduzido pela metade, aproximadamente, até que o elemento seja encontrado ou até que a lista não possa mais ser dividida.

Esse método foi batizado de método de busca algorítmica pelo fato de haver uma redução algorítmica de elementos a serem pesquisados, mas, como ocorre essa redução pela metade dos elementos da busca em cada comparação, esse método é conhecido como **método de busca binária** ou **busca logarítmica**.

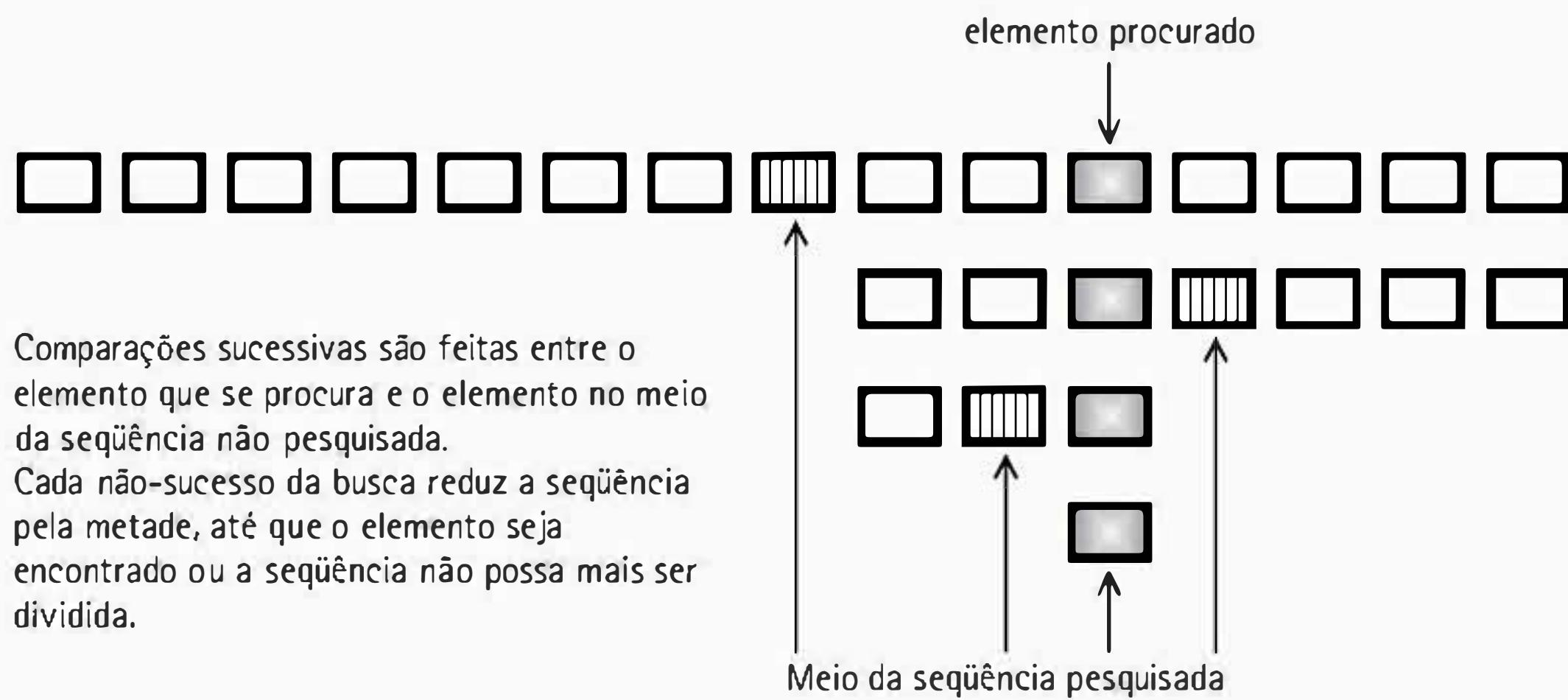
Para um conjunto de 15 elementos (como será mostrado na Figura 8.3), o método de busca linear realizaria, no mínimo, uma comparação e, no máximo, 15 comparações. No caso da busca binária, no mínimo ocorreria uma comparação e, no máximo, quatro comparações, o que geraria um ganho considerável, mais sensível para conjuntos de elementos muito grandes. Considere o exemplo abaixo, que demonstra isso.

Suponha que o elemento a ser localizado seja o 329, que será chamado de x :

500	178	2	487	158	47	35	78	329	215	19	25	214	38	77
-----	-----	---	-----	-----	----	----	----	-----	-----	----	----	-----	----	----

1º passo – Ordenar o conjunto:

2	19	25	35	38	47	77	78	158	178	214	215	329	487	500
---	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----



| FIGURA 8.3 | Busca binária

2º passo – Dividir o conjunto ao meio:

2	19	25	35	38	47	77	78	158	178	214	215	329	487	500
↑														

3º passo – Efetuar a comparação para verificar se o elemento procurado é igual ao elemento central, que será chamado de meio:

`x = meio`

Se o resultado for verdadeiro, a busca deverá ser encerrada. Caso contrário, serão executados os passos seguintes. No caso do nosso exemplo:

`329 = 78`, a resposta é falso.

4º passo – Efetuar a comparação para verificar se o elemento procurado é maior ou menor do que o elemento central:

`329 >= 78`, a resposta é verdadeiro.

5º passo – Proceder a uma nova divisão do conjunto que atenda à condição do 4º passo, isto é, se `x` for maior que `meio`, será dividido o conjunto da direita, senão, o conjunto da esquerda. No caso do exemplo, será utilizado o conjunto da direita.

158	178	214	215	329	487	500
↑						

Repetir os passos 4 e 5 até que o elemento seja encontrado.

329	487	500
↑		

EXEMPLO 8.8: Busca binária (ou logarítmica).

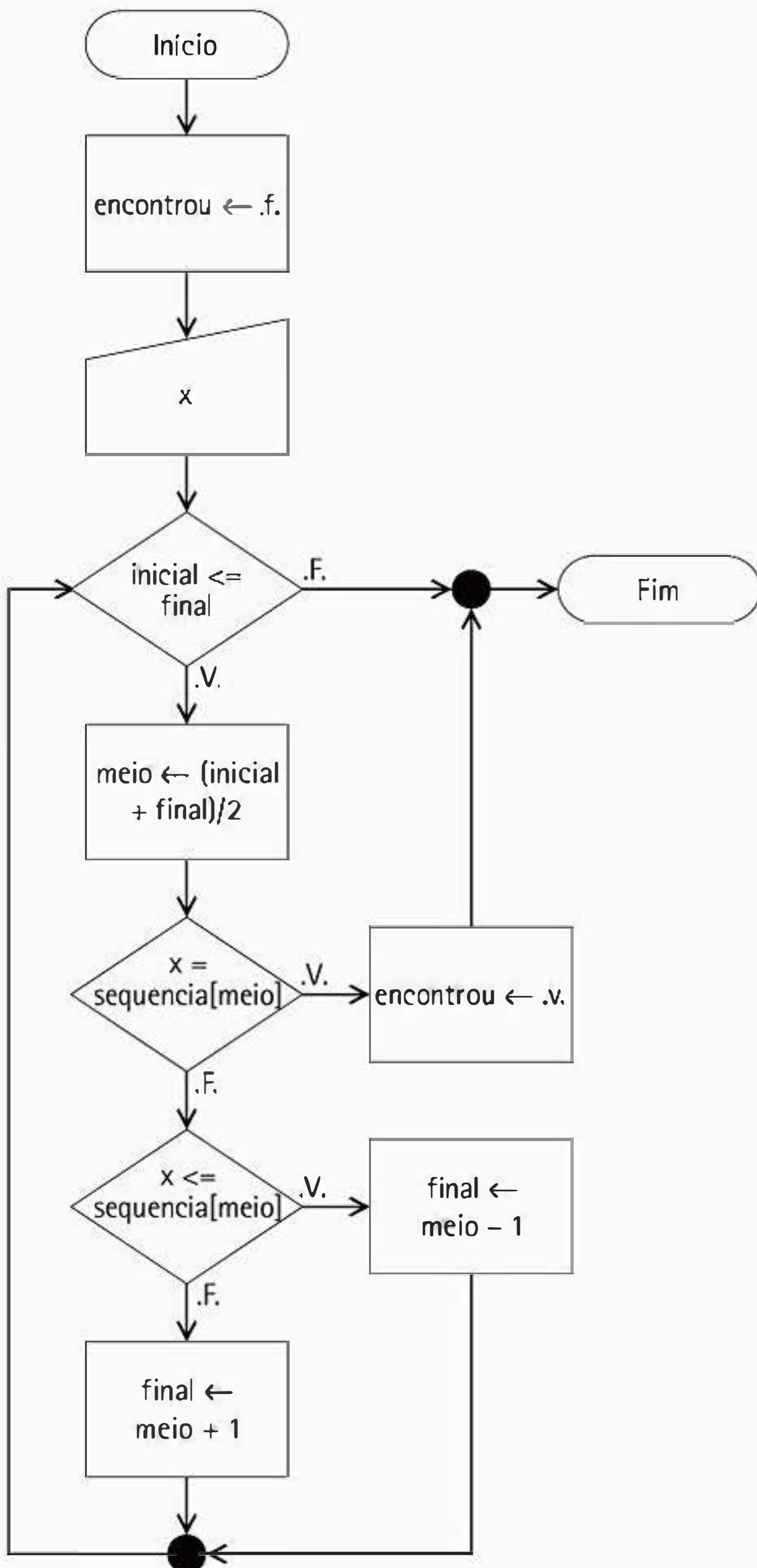
```

1. Algoritmo Exemplo8.8
2. Var
3.     sequencia : vetor de elementos           elemento a ser procurado
4.     x         : inteiro
5.     encontrou : logica
6.     inicial, meio, final: números inteiros      estes serão os índices
7. Início                                         da seqüência
8.     encontrou ← .f.
9.     Ler (x)
10.    Enquanto (inicial <= final) Faça
11.        meio ← (inicial + final) / 2
12.        Se (x = sequencia[meio])
13.            encontrou ← .v.
14.            <fim da busca>
15.        Senão
16.            Se (x <= sequencia[meio])
17.                final ← meio - 1
18.            Senão
19.                inicial ← meio + 1
20.            Fim-Se
21.        Fim-Se
22.    Fim-Enquanto
23. Fim.

```

O algoritmo de busca binária (Exemplo 8.8) é, na verdade, bem simples. Considerando uma seqüência qualquer, o índice do meio da seqüência é dado pela divisão por 2 da soma do primeiro com o último índice do conjunto que se deseja pesquisar (linha 11) — cuidado para garantir o resultado inteiro dessa divisão, uma vez que índices só podem ser números inteiros. Compara-se o elemento procurado ao elemento do meio da seqüência (linha 12) e, se forem iguais, a busca é encerrada (linha 14). Caso contrário, uma segunda comparação é realizada para determinar se o elemento procurado é maior ou menor que o elemento encontrado no meio (linha 16). Se o elemento é menor, significa que a segunda metade da seqüência pode ser ignorada para a próxima pesquisa, e isso pode ser alcançado alterando-se o valor do último índice do conjunto que se deseja pesquisar para o valor inferior do meio (linha 17). Caso contrário, isto é, caso o elemento seja maior que o elemento encontrado, então a primeira parte da seqüência pode ser ignorada, alterando-se o índice do início da busca para o valor imediatamente superior ao do meio (linha 19).

Isso será repetido até que o elemento seja encontrado ou até que a seqüência não possa mais ser dividida, o que ocorre, na prática, quando o índice do fim torna-se menor que o índice do início. Nesse caso, o elemento não se encontra na seqüência e retorna-se falso.

Fluxograma:

LEMBRE-SE: Para implementar estes algoritmos, as variáveis **inicial** e **final** devem receber algum valor!

O programa em Java para o Exemplo 8.8 segue exatamente a mesma lógica. No começo do algoritmo, o índice do início do conjunto a ser pesquisado será o índice do primeiro elemento da seqüência e o índice do fim será o índice do último elemento da

seqüência. Lembramos que o índice do primeiro elemento de um vetor, em Java, é 0 e, portanto, o último elemento terá índice igual à quantidade de elementos do vetor menos um.

Java:

```
1. class BuscaBinaria{  
2.     public static boolean binaria(int x, int numeros []) {  
3.         int inicio = 0, fim = numeros.length-1;  
4.         int meio;  
5.         while (inicio <= fim) {  
6.             meio = (inicio + fim) / 2;  
7.             if (x == numeros[meio])  
8.                 return true;  
9.             if (x < numeros[meio])  
10.                fim = meio - 1;  
11.            else  
12.                inicio = meio + 1;  
13.        }  
14.        return false;  
15.    }  
16. }
```

Esse programa apresenta apenas a criação de um método para busca de números inteiros. Seguindo-se as mesmas considerações de comparações dadas anteriormente, pode-se modificar ou ampliar essa classe para fazer pesquisa por meio do método de busca binária para outro tipo qualquer ou para strings.

8.3 EXERCÍCIOS PARA FIXAÇÃO

Para os problemas abaixo, faça o pseudocódigo, o fluxograma e o programa em Java.

1. Crie uma matriz com 20 números inteiros que deverão ser fornecidos aleatoriamente pelo usuário e:
 - a) coloque os elementos na ordem crescente;
 - b) coloque os elementos na ordem decrescente.
2. Crie uma matriz com nomes e ordene-a.
3. Elabore uma matriz com as seguintes informações, que deverão ser fornecidas pelo usuário: nome, sexo e idade, e:
 - a) ordene os elementos;
 - b) crie uma rotina para procurar um nome fornecido pelo usuário.
4. Reescreva o Exemplo 8.6 acrescentando outros métodos de busca linear para os demais tipos primitivos.

5. Reescreva o Exemplo 8.7 modificando o programa para que possua um menu de opções para entrada de dados (incluindo escolha de tipo), ordenação e busca.
6. Dada uma matriz A com N elementos do tipo real, elaborar um algoritmo que possibilite a busca binária de um valor qualquer fornecido pelo usuário.
7. Uma matriz X é composta pelos elementos do alfabeto e pelos numerais de 0 a 9. Escreva um algoritmo que seja capaz de localizar, pelo método binário, um caractere fornecido pelo usuário. Se esse caractere for uma letra o usuário poderá fornecê-la no formato maiúsculo ou minúsculo.
8. Dada uma tabela de horários de ônibus que fazem viagens para as diversas cidades do Estado de São Paulo, escreva um algoritmo que possibilite a localização dos horários de saída e de chegada quando se forneça o destino.

8.4 EXERCÍCIOS COMPLEMENTARES

1. Elabore uma matriz que armazene dados de 30 funcionários de uma empresa. Deverão ser considerados os campos: código funcional, nome, salário e data de admissão.
 - a) Preencha a matriz com dados fornecidos pelo usuário.
 - b) Ordene os elementos pelo campo código funcional.
 - c) Crie uma rotina para encontrar os dados de um funcionário pelo método binário.
2. Elabore uma matriz com três linhas e 20 colunas, que deverá ser preenchida com elementos sorteados aleatoriamente (utilize o método random), e:
 - a) faça a busca, pelo método binário, de um elemento sorteado;
 - b) faça a busca, pelo método seqüencial, de um elemento sorteado.

ACESSO A ARQUIVOS

- ▶ *Acesso a arquivos texto*
- ▶ *Operações de manipulação*
- ▶ *Exemplos e exercícios*



OBJETIVOS:

Trabalhar as técnicas para representação da criação e manipulação de arquivos texto, depois implementar esses algoritmos utilizando a ferramenta Java.

Até agora, todos os exemplos apresentados armazenavam as informações temporariamente na memória RAM do computador. Esse recurso é bastante utilizado durante a fase de aprendizado e também quando se está desenvolvendo o algoritmo para a resolução de um problema sem que haja, ainda, a preocupação com recursos de implementação como armazenamento de dados, interface e estética, entre outros.

As informações podem ser armazenadas em arquivos ou bancos de dados. Ambos são conjuntos de informações armazenadas em um meio magnético. A principal diferença entre eles está relacionada com a organização do armazenamento, o acesso e a recuperação dos dados. Os arquivos podem ser facilmente criados utilizando-se os recursos disponíveis no sistema operacional do computador, ao passo que, para a criação de um banco de dados, é necessário um software específico.

Neste livro, serão abordados apenas os arquivos, pois, para que fosse feita uma abordagem satisfatória acerca do armazenamento em bancos de dados, seria necessário o estudo dos conceitos de bancos de dados e das ferramentas para sua criação e gerenciamento.

NOTA:

Sistema operacional: software que é responsável pelo gerenciamento do hardware e do software da máquina, além de realizar a interface entre o usuário e o hardware.

9.1 O QUE É UM ARQUIVO?

Um **arquivo** é um local reservado para se guardar informações escritas. Um bom exemplo são os arquivos de aço utilizados para armazenar fichas contendo dados de clientes, produtos e pacientes, entre outros.

O arquivo de computador é uma maneira de armazenar informações em meios magnéticos, como por exemplo discos rígidos, discos flexíveis e CD-ROM, entre outros. Esses dados podem ser utilizados diversas vezes pelos programas associados ao arquivo.

Os arquivos podem armazenar imagens, textos ou sons. A título de exemplo, será demonstrado como armazenar informações-texto.

9.2 ARQUIVO-TEXTO

As informações de um **arquivo-texto** são organizadas em registros. Os registros, por sua vez, são organizados em campos e nos campos é que são ‘inseridas’ as informações. Suponha que, no exemplo abaixo, a Figura 1 represente a interface e a Figura 2 represente o arquivo-texto:

Nome: _____
Endereço: _____
CEP: _____ Tel.: _____

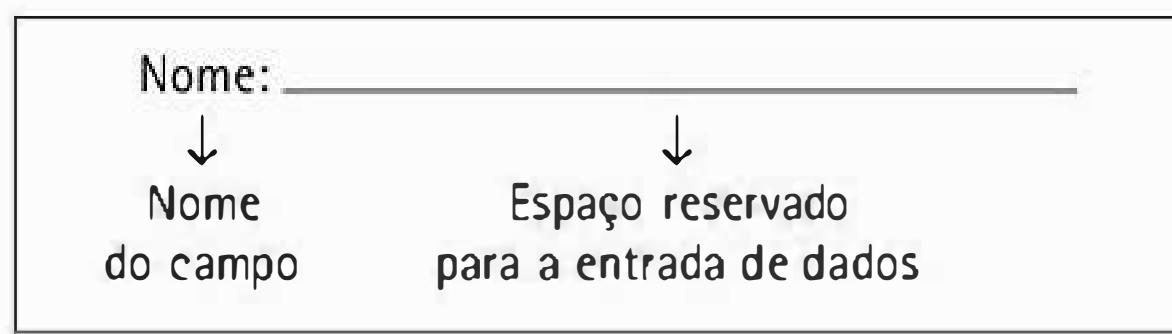
| FIGURA 9.1 | Interface

Nome	Endereço	CEP	Tel.
João Ninguém	Rua do Bosque, 10	08000102	67867766
Maria Bonita	Rua da Gruta, 247	09009904	31237788
José Filho Jr.	Av. Sul, 3.196	07989991	78966998

| FIGURA 9.2 | Arquivo-texto

- Na interface, são representados os nomes dos campos, que devem ser legíveis para o usuário, isto é, o usuário deve ler e entender o que deverá ser digitado nos espaços reservados para as informações.
- Na interface também deve existir um espaço para que o usuário faça a entrada dos dados, isto é, o local no qual serão digitadas ou selecionadas as informações.
- No programa, deve existir uma variável para cada informação que será digitada na interface.

- No arquivo, deve existir um campo para cada variável cuja informação deve ser armazenada.



No arquivo, cada linha representa um registro. Um registro é um conjunto de campos, isto é, um conjunto de informações sobre um determinado assunto. Cada campo recebe dados que são armazenados no arquivo por meio da associação da variável da interface com o campo do arquivo.

NOTA:

Os tipos de dados das variáveis e campos são os mesmos estudados no Capítulo 3.

Os registros são estruturas de dados compostos por um conjunto de campos definidos.

Registros	Campos			
	Nome	Endereço	CEP	Tel.
João Ninguém	Rua do Bosque, 10	08000102	67867766	
Maria Bonita	Rua da Gruta, 247	09009904	31237788	
José Jr. Filho	Av. Sul, 3.196	07989991	78966998	

↓
Dados

9.3 TIPOS DE ARQUIVO QUANTO ÀS FORMAS DE ACESSO

Os arquivos-texto podem ser diferenciados quanto à forma de acesso aos dados: arquivos de acesso seqüencial e arquivos de acesso aleatório ou randômico.

9.3.1 ARQUIVOS SEQÜENCIAIS

Os **arquivos seqüenciais** armazenam informações em formato ASCII e as informações são armazenadas na ordem em que são digitadas. Os arquivos seqüenciais apresentam alguns inconvenientes:

- As informações são lidas na mesma ordem em que foram inseridas, isto é, em seqüência.
- É necessário que o arquivo todo seja percorrido até que a informação seja localizada.
- As informações não podem ser alteradas diretamente no arquivo. Para se alterar algum dado, é necessário copiar para um novo arquivo todas as informações do arquivo anterior, já com as alterações.

- Esse tipo de arquivo não é recomendado para trabalhos com grande volume de informações, pois é lento.
- Não é possível abri-lo para leitura e escrita.

NOTA:

ASCII — American Standard Code for Information Interchange, que significa Código Padronizado Americano para Intercâmbio de Informações. A tabela ASCII possui 256 combinações de 8 bits, que representam caracteres.

9.3.2 ARQUIVOS DE ACESSO ALEATÓRIO

Os **arquivos de acesso aleatório** ou randômico também armazenam as informações em formato ASCII. Cada registro é armazenado em uma posição específica. Com isso, as informações podem ser lidas independentemente da ordem em que foram inseridas.

9.4 OPERAÇÕES DE MANIPULAÇÃO DE ARQUIVOS

As informações dos arquivos, por estarem armazenadas em um dispositivo físico, podem ser manipuladas, isto é, o arquivo pode ser atualizado ou simplesmente consultado.

Para a manipulação dos arquivos, existem quatro operações básicas que podem ser realizadas:

- Inserção de dados: trata-se da inclusão de novos registros.
- Consulta aos dados: trata-se da leitura dos dados já armazenados.
- Alteração dos dados: trata-se da possibilidade de alterar de um ou mais campos do conjunto.
- Exclusão de dados: trata-se da eliminação de registros.

9.4.1 REPRESENTAÇÃO DA MANIPULAÇÃO DE ARQUIVOS SEQÜENCIAIS

Para que seja possível manipular arquivos seqüenciais, será necessário:

- Declarar o registro e o arquivo.
- Declarar as variáveis de arquivo e registro.
- Abrir o arquivo.
- Fechar o arquivo.

Esses passos são utilizados para qualquer operação de manipulação de arquivos e serão explicados a seguir, no Exemplo 9.1 — Operação de inclusão.

Operação de inclusão — arquivo seqüencial

EXEMPLO 9.1: Construção de uma agenda que armazene nomes, endereços e telefones.

Pseudocódigo:

```

1. Algoritmo Exemplo_9.1
2. Var
3.     Tipo reg_agenda = registro
4.             Nome: caracter
5.             End: caracter
6.             Tel: caracter
7.         Fim_registro
8.     Tipo arq_agenda: arquivo seqüencial de reg_agenda
9.     Auxiliar: reg_agenda
10.    Agenda: arq_agenda
11. Início
12.    Abrir (Agenda)
13.    Repita
14.        Avançar (Agenda)
15.        Até EOF (Agenda)
16.        Ler (Auxiliar.Nome, Auxiliar.End, Auxiliar.Tel)
17.        Armazenar (Agenda, Auxiliar)
18.        Fechar (Agenda)
19.    Fim.

```

NOTA: *EOF — end of file (fim de arquivo).*

Nos algoritmos, deve-se, primeiramente, declarar uma estrutura do tipo **registro**, conforme aprendido no Capítulo 6, com todos os campos cujas informações pretende-se armazenar, como no trecho das linhas 3 a 7 do Exemplo 9.1:

```

Tipo reg_agenda = registro
    Nome: caracter
    End: caracter
    Tel: caracter
Fim_registro

```

É necessário também declarar um identificador do tipo **arquivo**, mencionando o tipo de arquivo que será utilizado. Esse identificador é associado ao arquivo que será formado pelos registros de **reg_agenda**.

As variáveis **Auxiliar** e **Agenda** são variáveis de **registro** e de **arquivo**, respectivamente.

Para que seja possível a manipulação do arquivo, esse deve ser aberto:

Abrir (nome da variável de arquivo)

Será disponibilizado o primeiro registro armazenado. Para acessar os próximos registros, utiliza-se a instrução:

Avançar (nome da variável de arquivo)

No caso de se desejar que o arquivo seja posicionado no último registro, utiliza-se uma estrutura de repetição que provoque o avanço pelos registros até o final do arquivo:

Repita

 Avançar (nome da variável de arquivo)

 Até EOF (nome da variável de arquivo)

NOTA:

Por se tratar de um arquivo seqüencial, para se chegar ao último registro, percorre-se o arquivo todo, passando por todos os registros armazenados, como, por exemplo, em uma fita cassete.

Para processar a inclusão de um registro, é necessário que seus campos sejam preenchidos na mesma ordem e com os mesmos campos do arquivo. Por isso, a variável de registro é declarada com o tipo da estrutura declarada para o registro do arquivo. No caso do Exemplo 9.1:

- Declaração da estrutura de dados do tipo registro:

```
Var Tipo reg_agenda = registro
    Nome: caracter
    End: caracter
    Tel: caracter
Fim_registro●
```

- Declaração da variável do tipo registro, que terá o mesmo formato da estrutura de dados criada para o registro, isto é, Auxiliar.Nome, Auxiliar.End, Auxiliar.Tel:

Auxiliar: reg_agenda

- Preenchimento dos campos:

Ler (Auxiliar.Nome, Auxiliar.End, Auxiliar.Tel)

Depois de representar o preenchimento dos campos, será necessário representar o armazenamento do conteúdo no arquivo. Para isso utiliza-se a instrução:

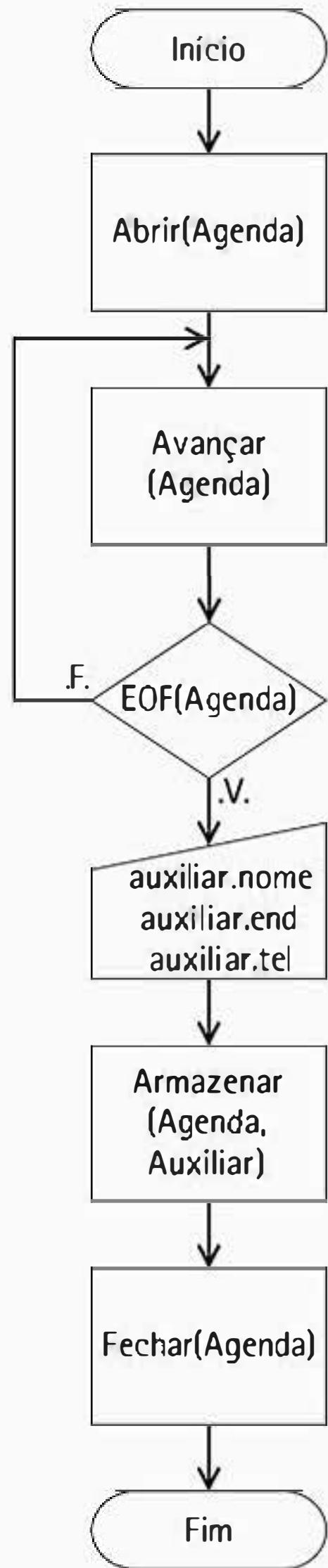
Armazenar (nome da variável de arquivo, nome da variável de registro)

Armazenar (Agenda, Auxiliar)

A variável de arquivo Agenda receberá o conteúdo da variável de registro Auxiliar.

Por último, o arquivo deve ser fechado:

Fstrar (nome da variável de arquivo)

Fluxograma:**Java:**

```
1. class regAgenda{
2.     private String nome;
3.     private String end;
4.     private String tel;
5.
6.     public regAgenda (String nom, String ender, String telef){
7.         nome = nom;
8.         end = ender;
9.         tel = telef;
10.    }
11.    public String mostraNome () {
```

```

12.         return nome;
13.     }
14.     public String mostraEnd (){
15.         return end;
16.     }
17.     public String mostraTel (){
18.         return tel;
19.     }
20. }
```

Para representar uma estrutura do tipo registro em Java, utilizou-se uma classe `regAgenda` que possui como atributos: `nome`, `end` e `tel`, conforme trecho de código acima. Essa forma de representação é necessária, uma vez que o Java trabalha com objetos, conforme citado no Capítulo 4. Assim, cada entrada em `regAgenda` corresponde a um novo registro e refere-se a um novo objeto da classe `regAgenda`, que possui características próprias, ou seja, um nome, um endereço e um telefone que o distinguem dos demais objetos existentes.

A declaração da classe correspondente ao registro é feita nas linhas de 1 a 4. Nas linhas de 6 a 9 é definido o método construtor, responsável pela instanciação de novos objetos da classe ou, para o nosso caso, de novas entradas de registro em `regAgenda`.

Para exibir os atributos de cada novo objeto instanciado, criaram-se os métodos `mostraNome()`, `mostraEnd()` e `mostraTel()`, que retornamos respectivos dados.

O algoritmo codificado em Java relativo ao Exemplo 9.1 e que utiliza a classe `regAgenda` para definição das entradas de novos registros na agenda é apresentado a seguir.

CUIDADO!

Java é uma linguagem case-sensitive. As referências às variáveis e métodos devem ser feitas exatamente da forma que estes foram declarados, considerando as letras maiúsculas e minúsculas.

```

1. import java.io.*;
2. class Exemplo91{
3.     public static void main(String[] args){
4.         try{
5.             BufferedReader entrada;
6.             entrada = new BufferedReader (new InputStreamReader
7.             (System.in));
8.             BufferedWriter saida;
9.             saida = new BufferedWriter (new FileWriter
10.            ("Agenda.txt", true));
11.             System.out.println ("Digite o nome");
12.             String Nome = entrada.readLine();
13.             System.out.println ("Digite o endereço");
14.             String Endereco = entrada.readLine();
15.             System.out.println ("Digite o telefone");
```

```

14.         String Telefone = entrada.readLine();
15.         regAgenda regAgl = new regAgenda (Nome, Endereco,
   Telefone);
16.         saida.write (regAgl.mostraNome() + "\t");
17.         saida.write (regAgl.mostraEnd () + "\t");
18.         saida.write (regAgl.mostraTel () + "\n");
19.         saida.flush ();
20.         saida.close ();
21.     }catch (Exception e) {
22.         System.out.println ("Erro de gravacao");
23.     }
24. }
25. }
```

Para ler a entrada dos dados via teclado, declarou-se uma variável `entrada`, que é um `contêiner` que recebe os caracteres de entrada digitados via teclado (linhas 5 e 6). Seguindo a mesma filosofia da entrada de dados, utilizam-se classes (e objetos) de apoio definidos na linguagem Java para trabalhar o acesso e gravação de arquivos.

Nas linhas 7 e 8 está a declaração da variável (objeto) que será o `contêiner` do arquivo. Essa declaração é muito semelhante à declaração das variáveis que receberão a entrada de dados, mas, em vez de se utilizar um objeto do tipo `InputStreamReader`, que define uma entrada de caracteres, utiliza-se um objeto do tipo `FileWriter` (linha 8), que define uma saída para o arquivo `Agenda.txt`. Um detalhe particular deve ser observado nessa declaração: na linha 8, quando se define o nome do arquivo destinado ao armazenamento dos dados, no trecho `FileWriter ("Agenda.txt", true)`, tem-se, na verdade, uma chamada de um método com passagem dos parâmetros `"Agenda.txt"` e `true`. O primeiro é o nome do arquivo, conforme já foi dito, e o segundo significa que o arquivo será acessado com a condição `append = true`, ou seja, os dados gravados serão sempre inseridos ao final do arquivo. Vale ressaltar ainda que, caso o arquivo não exista, ele será criado automaticamente.

O processo de entrada dos dados é executado a partir das instruções das linhas 9 a 14, segundo as quais os caracteres lidos a partir do teclado serão transferidos para as respectivas variáveis de armazenamento: `Nome`, `Endereco` e `Telefone`.

Na linha 15, ocorre a chamada do método construtor da classe, `regAgenda`, com a respectiva passagem dos parâmetros `Nome`, `Endereco` e `Telefone`, instanciando-se um novo objeto `regAgl` da classe `regAgenda`. No trecho de código anterior (classe `regAgenda`), pode-se verificar a declaração do método nas linhas 6 a 9.

A saída dos dados é feita por meio do código nas linhas 16 a 18, com a chamada dos métodos que retornam os atributos do novo objeto criado. Por exemplo, em

```
saida.write (regAgl.mostraNome() + "\t"),
```

temos a chamada do método que retorna o atributo `nome` do objeto `regAgl` e a transferência do resultado para a saída. O atributo `"\t"` associado ao nome indica que, após o dado, será gravado também um espaço de tabulação como separador.

Ao final da transferência do último dado, também é passado o caractere de controle “\n” (nova linha), utilizado como separador, indicando o final do registro. O motivo da utilização desses separadores será visto nos exemplos posteriores.

Nas linhas 19 e 20, são chamados os métodos `saída.flush()` e `saída.close()`, que fazem a transferência definitiva dos dados da memória para o arquivo e o fechamento do arquivo, respectivamente.

NOTA:

Pode-se perguntar por que os dados obtidos da entrada via teclado não são transferidos diretamente para o arquivo, uma vez que eles estariam disponíveis sem a necessidade de instanciar um objeto da classe regAgenda. O motivo é a necessidade de satisfazer os princípios da orientação a objetos, segundo os quais cada novo dado ou registro é um objeto e assim deve ser tratado. Como será visto posteriormente, a manipulação dos atributos dos objetos deve seguir esses princípios, garantindo-se que alterações nesses atributos somente poderão ser feitas mediante a chamada de métodos, que permitirão ou não as alterações. Os atributos de um objeto definem sua estrutura, e as operações, seu comportamento. Os métodos implementam essas operações que permitem alterações nos objetos e na sua estrutura, preservando sua integridade (princípio do encapsulamento discutido no Capítulo 4).

Operação de consulta — arquivo seqüencial

A operação de consulta pode ser realizada de duas maneiras:

- saltando manualmente de um registro para o outro: nesse caso, o usuário visualizará todos os registros até que se chegue ao registro desejado;
 - saltando automaticamente para o registro desejado: nesse caso, utiliza-se uma variável que recebe a informação a ser encontrada no arquivo e, por meio de uma estrutura de repetição, é provocado o avanço pelos registros até que seja encontrado o registro desejado.

NOTAI

Quando se trabalha com arquivos seqüenciais, em ambos os casos todos os registros são percorridos até que se chegue ao registro desejado.

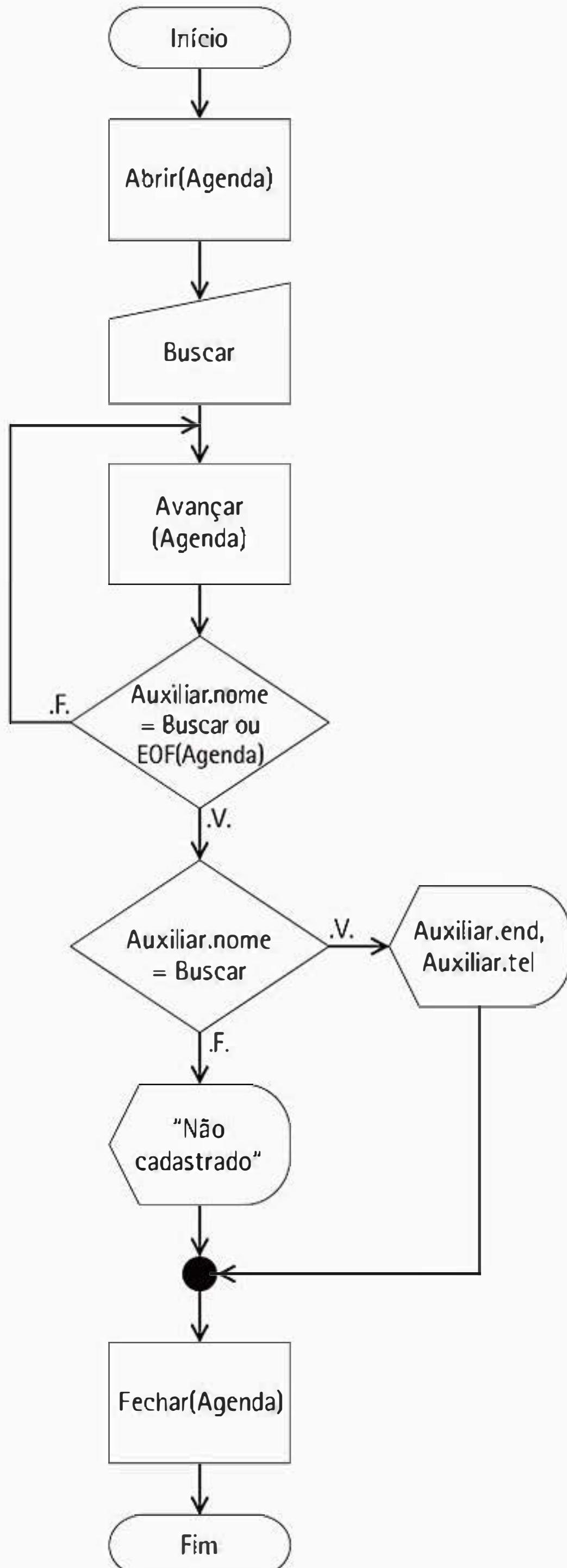
No exemplo a seguir, será utilizado o recurso de consulta automática. Para isso, será declarada a variável `Buscar`, que receberá o nome a ser consultado.

EXEMPLO 9.2: Busca sequencial em arquivo.

- ```
1. Algoritmo Exemplo_9.2
2. Var tipo reg_agenda = registro
3. Nome: caracter
4. End: caracter
5. Tel: caracter
6. Fim registro
```

```
7. Tipo arq_agenda: arquivo seqüencial de agenda
8. Auxiliar: reg_agenda
9. Agenda: arq_agenda
10. Buscar: caracter
11. Inicio
12. Abrir (Agenda)
13. Ler(Buscar)
14. Repita
15. Avançar (Agenda)
16. Até (Auxiliar.Nome = Buscar) ou (EOF(Agenda))
17. Se (Auxiliar.Nome = Buscar) Então
18. Mostrar(Auxiliar.End, Auxiliar.Tel)
19. Senão
20. Mostrar("Não cadastrado")
21. Fim-Se
22. Fechar (Agenda)
23. Fim.
```

Nesse exemplo, foi utilizada a variável **Buscar**, que recebeu o nome a ser consultado. A estrutura de repetição **Repita** provoca o avanço pelos registros do arquivo até que se encontre o nome desejado ou o final do arquivo. Se tivermos sucesso na consulta, isto é, se o nome desejado for encontrado, então o programa exibirá o endereço e o telefone.

**Fluxograma:**

**Java:**

```
1. import java.io.*;
2. class Exemplo92{
3. public static void main(String[] args) {
4. try{
5. BufferedReader entrada;
6. entrada = new BufferedReader (new InputStreamReader
7. (System.in));
8. BufferedReader argentrada;
9. argentrada = new BufferedReader (new FileReader
("Agenda.txt"));
10. System.out.println ("Digite o nome");
11. String Nome = entrada.readLine();
12. StringBuffer memoria = new StringBuffer();
13. String linha;
14. while ((linha = argentrada.readLine()) != null) {
15. memoria.append(linha + "\n");
16. }
17. int inicio = -1;
18. inicio = memoria.indexOf (Nome);
19. if (inicio != -1){
20. int fim = memoria.indexOf ("\n", inicio);
21. char [] destino = new char [fim - inicio];
22. memoria.getChars (inicio, fim, destino, 0);
23. linha = "";
24. for (int i = 0; i < fim - inicio; i++){
25. linha += destino [i];
26. }
27. System.out.println (linha);
28. }else{
29. System.out.println ("Item nao encontrado");
30. }
31. entrada.close ();
32. } catch (FileNotFoundException erro){
33. System.out.println ("Arquivo nao encontrado!");
34. } catch (Exception erro){
35. System.out.println ("Erro de Leitura!");
36. }
37. }
```

Na implementação desse exemplo, cria-se uma estrutura chamada `memoria`, que é uma variável de memória do tipo `StringBuffer`. Em Java, esse tipo de variável é semelhante ao tipo `String`, mas permite uma estrutura de trabalho mais avançada como, por exemplo, `append`, que adiciona um novo conteúdo ao conteúdo já existente, sem perda de dados.

A linha 13 define a condicional de uma estrutura de armazenamento de dados que se repetirá enquanto houver dados a serem lidos, isto é, lerá todos os dados até o fim do arquivo. Essa condicional é equivalente à determinação de EOF (*end of file*).

A busca é feita na variável `memoria`, com o método `indexOf`, que recebe `Nome` como parâmetro, dado obtido pela entrada de dados via teclado na linha 10. Esse método retorna a posição onde se inicia o caractere pesquisado. As posições são numeradas a partir de 0, significando que o teste de condição na linha 18 verifica se o valor atribuído inicialmente à variável `inicio` foi alterado. Em caso afirmativo, ou seja, se `inicio` é diferente de `-1`, executam-se os comandos internos do laço.

Para que seja possível obter a seqüência de caracteres correspondentes ao registro completo, é necessário determinar o seu final. Isso é feito por meio do comando na linha 19, que busca o caractere "`\n`" (nova linha) imediatamente após o primeiro caractere encontrado. É importante observar que o método `indexOf` é chamado novamente, porém, dessa vez, são passados dois parâmetros: "`\n`" e `inicio`, para que a nova busca ocorra a partir da posição em que foi encontrado o primeiro caractere do item pesquisado.

Na linha 21, utiliza-se o método `getChars` para se obter os caracteres correspondentes ao registro, passando-se os parâmetros `inicio` e `fim` como delimitadores. A string obtida é armazenada na própria variável `linha`, que será utilizada para exibição dos dados (linha 26).

Foram feitos dois tratamentos de erro nas linhas 31 e 33, para determinar problemas quando o arquivo não for encontrado ou quando ocorrerem outros erros genéricos, respectivamente.

#### **EXEMPLO 9.3: Operação de alteração — arquivo seqüencial.**

```

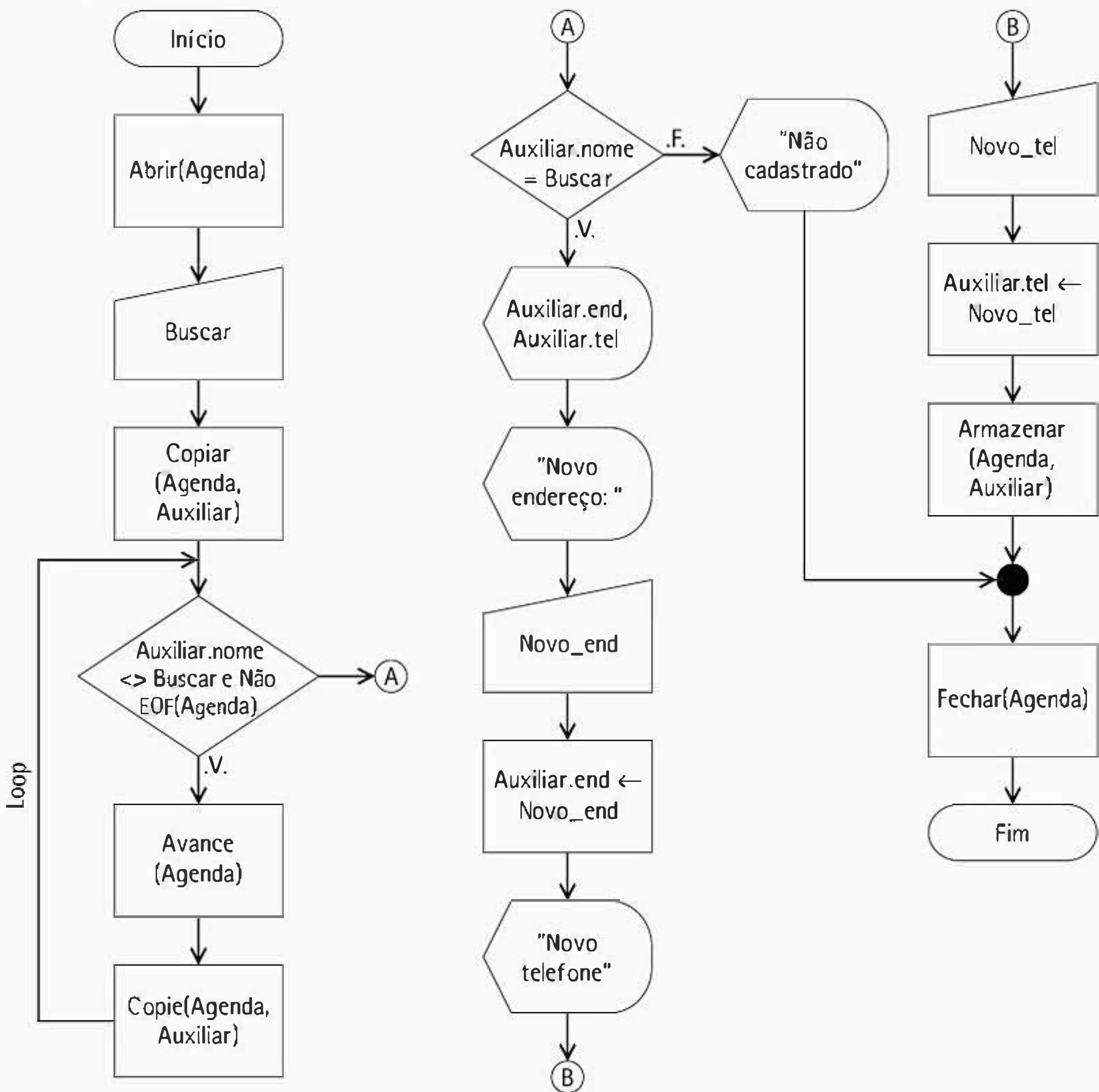
1. Algoritmo Exemplo_9.3
2. Var tipo reg_agenda = registro
3. Nome: caracter
4. End: caracter
5. Tel: caracter
6. Fim_registro
7. Tipo arq_agenda: arquivo seqüencial de agenda
8. Auxiliar: reg_agenda
9. Agenda: arq_agenda
10. Buscar: caracter
11. Novo_end: caracter
12. Novo_tel: caracter
13. Inicio
14. Abrir (Agenda)
15. Ler (Buscar)
16. Copiar (Agenda, Auxiliar)
17. Enquanto (Auxiliar.Nome <> Buscar) e (Não EOF(Agenda)) Faça
18. Avançar (Agenda)

```

```
19. Copiar(Agenda, Auxiliar)
20. Fim-Enquanto
21. Se (Auxiliar.Nome = Buscar) Então
22. Início
23. Mostrar(Auxiliar.End, Auxiliar.Tel)
24. Mostrar("Novo endereço: ") Ler(Novo_end)
25. Auxiliar.end ← Novo_end
26. Mostrar("Novo telefone: ") Ler (Novo_tel)
27. Auxiliar.tel ← Novo_tel
28. Armazenar(Agenda, Auxiliar)
29. Fim
30. Senão
31. Mostrar("Não cadastrado")
32. Fim-Se
33. Fechar (Agenda)
34. Fim.
```

No Algoritmo 9.3 é feita uma consulta similar à do Algoritmo 9.2. Se o registro for encontrado, é recomendado que seja exibido antes da alteração. Para a alteração, devem-se declarar as variáveis que receberão os novos valores, preencher os novos valores e depois atribuir esses valores às variáveis de registro:

```
Mostrar("Novo endereço: "); Ler(Novo_end)
Auxiliar.end ← Novo_end
```

**Fluxograma:****Java:**

```

1. public class regAgenda{
2. private String nome;
3. private String end;
4. private String tel;
5.
6. public regAgenda (String nom, String ender, String telef){
7. nome = nom;
8. end = ender;
9. tel = telef;
10. }
11. public String mostraNome (){
12. return nome;
13. }

```

```
14. public String mostraEnd (){
15. return end;
16. }
17. public String mostraTel (){
18. return tel;
19. }
20. public void alteraEnd (String novoEnd) {
21. end = novoEnd;
22. }
23. public void alteraTel (String novoTel) {
24. tel = novoTel;
25. }
26. }
```

A classe `regAgenda` sofreu alterações para suportar as novas necessidades do algoritmo. Foram incluídos os métodos `alteraEnd` e `alteraTel` (linhas 20 a 25) para que os objetos permitam alterações no endereço e no número do telefone. Assim, para que seja possível a alteração desses atributos, é necessário invocar (ou chamar) esses métodos, como será feito no trecho de código a seguir.

```
1. import java.io.*;
2. class Exemplo93{
3. static StringBuffer memoria = new StringBuffer();
4. public static void main(String[] args){
5. try{
6. BufferedReader entrada;
7. entrada = new BufferedReader (new InputStreamReader
(System.in));
8. BufferedReader argentrada;
9. argentrada = new BufferedReader (new FileReader
("Agenda.txt"));
10. System.out.println ("Digite o nome");
11. String Nome = entrada.readLine();
12. String Endereco = "";
13. String Telefone = "";
14. String linha = "";
15. while ((linha = argentrada.readLine()) != null) {
16. memoria.append (linha + "\n");
17. }
18. int inicio = -1;
19. inicio = memoria.indexOf (Nome);
20. if (inicio != -1){
21. int ultimo = memoria.indexOf ("\t", inicio);
22. Nome = Ler (inicio, ultimo);
23. int primeiro = ultimo + 1;
24. ultimo = memoria.indexOf ("\t", primeiro);
25. Endereco = Ler (primeiro, ultimo);
```

```
26. primeiro = ultimo + 1;
27. int fim = memoria.indexOf ("\n", primeiro);
28. Telefone = Ler (primeiro, fim);
29. regAgenda regAg1 = new regAgenda (Nome, Endereco,
 Telefone);
30. System.out.println ("Endereco: " +
 regAg1.mostraEnd () +
 "Telefone: " + regAg1.mostraTel ());
31. System.out.println ("Entre com novo endereco");
32. Endereco = entrada.readLine ();
33. regAg1.alteraEnd (Endereco);
34. System.out.println ("Entre com novo telefone");
35. Telefone = entrada.readLine ();
36. regAg1.alteraTel (Telefone);
37. memoria.replace (inicio, fim, regAg1.mostraNome () +
 "\t" +
 regAg1.mostraEnd () + "\t" + regAg1.mostraTel ());
38. gravar ();
39. argentrada.close ();
40. } else{
41. System.out.println ("Item nao encontrado");
42. }
43. argentrada.close ();
44. } catch (FileNotFoundException erro){
45. System.out.println ("Arquivo nao encontrado!");
46. } catch (Exception erro){
47. System.out.println ("Erro de Leitura!");
48. }
49. }
50. }
51. }
52. public static String ler (int primeiro, int ultimo){
53. String dados = "";
54. char [] destino = new char [ultimo - primeiro];
55. memoria.getChars (primeiro, ultimo, destino, 0);
56. for (int i = 0; i < destino.length; i++){
57. dados += destino [i];
58. }
59. return dados;
60. }
61. public static void gravar (){
62. try{
63. BufferedWriter saida;
64. saida = new BufferedWriter (new FileWriter
("Agenda.txt"));
65. saida.write (memoria.toString ());
66. saida.flush ();
67. saida.close ();
68. } catch (Exception erro){
69. System.out.println ("Erro de gravacao!");
```

```

70. }
71. }
72. }
```

**ATENÇÃO!**

**Observe que, para a execução do programa do Exemplo 9.3, o arquivo Agenda.txt deverá conter pelo menos um registro de entrada. Portanto, é necessário executar o programa do Exemplo 9.1 para a inclusão dos registros desejados.**

Os mesmos recursos de entrada de dados e busca em arquivo do Exemplo 9.2 foram utilizados. Porém, para obtermos os dados dos diversos campos separadamente, buscamos “\t” (tabulação) após a busca para localizar a primeira posição do nome e assim obter os caracteres correspondentes somente ao nome. Por exemplo, suponha que fosse incluído o seguinte registro:

Gilberto da Silva      Rua A n 203 apto 102      25834911

Para obtermos somente o campo nome (Gilberto da Silva), efetuamos a busca passando para o método `indexOf` os parâmetros “\t” e `inicio`, que correspondem respectivamente à tabulação e à posição em que foi encontrado o primeiro caractere do nome (variável `inicio`). Para obtermos os caracteres referentes ao campo nome, utilizamos o método `ler`, fornecendo como parâmetros as posições de `inicio` e `ultimo`, correspondentes ao primeiro e ao último caracteres do nome.

Observe também que foram incluídas as variáveis `primeiro` e `ultimo` em relação ao Exemplo 9.2. Essas variáveis cumprem o papel de obter as posições de início e fim de cada campo, visto que precisamos preservar o valor das variáveis `inicio` e `fim`, conforme explicaremos a seguir.

As variáveis `inicio` e `fim` se prestam a guardar as posições de início e fim do registro e foram utilizadas para invocar o método `replace` na linha 38, que faz a atualização do contêiner dos dados que estão na memória. Para tanto, é necessário identificar corretamente as posições e os dados que serão substituídos. Note que os parâmetros passados para o método são: `inicio`, `fim` e uma string composta pela concatenação `regAg1.mostraNome () + "\t" + regAg1.mostraEnd () + "\t" + regAg1.mostraTel ()`. Usando o exemplo citado acima e supondo que o endereço de Gilberto da Silva fosse alterado para Rua Alfazema n 203 apto 102, a string resultante que seria passada para atualização seria:

Gilberto da Silva      Rua Alfazema n 203 apto 102      25834911

O método `ler`, implementado a partir da linha 52, tem a finalidade de obter e retornar uma string de caracteres (variável `dados`), correspondente às posições fornecidas (`primeiro` e `ultimo`), de forma similar à utilizada no exemplo anterior.

Também foi implementado um método chamado `gravar`, cujo código encontra-se a partir da linha 61. Esse método usa os mesmos princípios apresentados no Exem-

plo 9.1. Note que ele não recebe nem retorna nenhum valor, executando a transferência dos dados do contêiner memória para o arquivo.

**EXEMPLO 9.4:** Operação de exclusão — arquivo seqüencial.

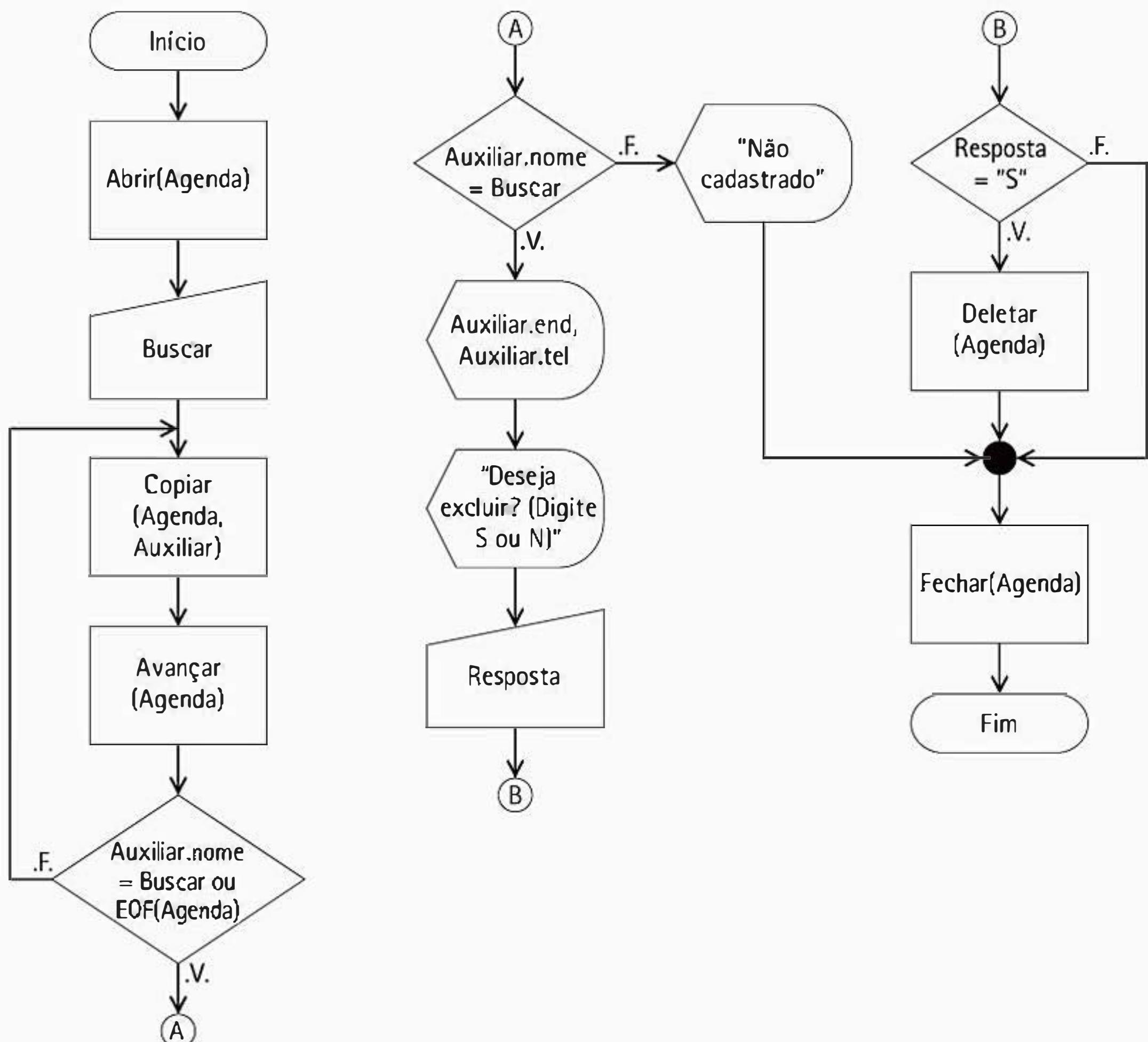
```

1. Algoritmo exemplo_9.4
2. Var tipo reg_agenda = registro
3. Nome: caracter
4. End: caracter
5. Tel: caracter
6. Fim_registro
7. Tipo arq_agenda: arquivo seqüencial de agenda
8. Auxiliar: reg_agenda
9. Agenda: arq_agenda
10. Buscar: caracter
11. Resposta: caracter
12. Inicio
13. Abrir (Agenda)
14. Ler(Buscar)
15. Repita
16. Copiar(Agenda,Auxiliar)
17. Avançar(Agenda)
18. Até (Auxiliar.Nome = Buscar) ou (EOF(Agenda))
19. Se (Auxiliar.Nome = Buscar) Então
20. Início
21. Mostrar(Auxiliar.End, Auxiliar.Tel)
22. Mostrar("Deseja Excluir? (Digite S ou N) ") Ler(resposta)
23. Se (resposta = "S") Então
24. Deletar(agenda)
25. Fim-Se
26. Fim
27. Senão
28. Mostrar("Não cadastrado")
29. Fim-Se
30. Fechar (Agenda)
31. Fim.

```

Para facilitar o processo de exclusão, é recomendado que seja feita uma busca automática, conforme visto anteriormente no Algoritmo 9.2. A exclusão será feita com a instrução **Deletar** (nome da variável de arquivo).

**NOTA:** *Uma vez deletado, o registro não pode ser recuperado.*

**Fluxograma:****Java:**

```

1. import java.io.*;
2. class Exemplo94{
3. static StringBuffer memoria = new StringBuffer();
4. public static void main(String[] args){
5. try{
6. BufferedReader entrada;
7. entrada = new BufferedReader (new InputStreamReader
(System.in));
8. BufferedReader arqentrada;
9. arqentrada = new BufferedReader (new FileReader
("Agenda.txt"));
10. System.out.println ("Digite o nome");
11. String Nome = entrada.readLine();
12. String linha = "";

```

```
13. while ((linha = arqentrada.readLine()) != null) {
14. memoria.append (linha + "\n");
15. }
16. int inicio = -1;
17. inicio = memoria.indexOf (Nome);
18. if (inicio != -1){
19. int fim = memoria.indexOf ("\n", inicio);
20. linha = ler (inicio, fim);
21. System.out.println ("Deseja excluir? Digite S ou N");
22. System.out.println (linha);
23. String resp = entrada.readLine ();
24. if (resp.equalsIgnoreCase ("S")){
25. memoria.delete (inicio, fim + 1);
26. System.out.println ("Registro excluido.");
27. }
28. gravar ();
29. }else{
30. System.out.println ("Item nao encontrado");
31. }
32. arqentrada.close ();
33. } catch (FileNotFoundException erro){
34. System.out.println ("Arquivo nao encontrado!");
35. } catch (Exception erro){
36. System.out.println ("Erro de Leitura!");
37. }
38. }
39. public static String ler (int primeiro, int ultimo){
40. String dados = "";
41. char [] destino = new char [ultimo - primeiro];
42. memoria.getChars (primeiro, ultimo, destino, 0);
43. for (int i = 0; i < destino.length; i++){
44. dados += destino [i];
45. }
46. return dados;
47. }
48. public static void gravar (){
49. try{
50. BufferedWriter saida;
51. saida = new BufferedWriter (new FileWriter
("Agenda.txt"));
52. saida.write (memoria.toString ());
53. saida.flush ();
54. saida.close ();
55. } catch (Exception erro){
56. System.out.println ("Erro de gravacao!");
57. }
58. }
59. }
```

Poucas modificações foram necessárias em relação ao código do Exemplo 9.3 para implementar a exclusão de registro. Utilizou-se para tanto uma `String resp` que recebe a entrada feita pelo teclado (linha 23), o que permite a interação com o usuário, pedindo-se uma confirmação da exclusão. Observe a particularidade da expressão que avalia a resposta. Usa-se `equalsIgnoreCase`, um método da classe `String`, para verificar a condição de igualdade desconsiderando maiúsculas e minúsculas. Atitude bastante salutar, pois o usuário pode não perceber a condição do teclado no momento (Caps Lock ativado ou não) e não obter sucesso no processo. Qualquer outra tecla digitada nesse momento invalida a exclusão. A exclusão ocorre no trecho de código na linha 25 (exclusão dos caracteres entre as posições `inicio` e `fim + 1`) somente no container `memoria`. A atualização do arquivo ocorre somente quando é chamado o método `gravar`. Note ainda que foi utilizado `fim + 1`, pois a posição `fim` correspondente ao final do registro não considera o caractere `\n`.

#### 9.4.2 REPRESENTAÇÃO DA MANIPULAÇÃO DE ARQUIVOS DE ACESSO ALEATÓRIO

As informações de um arquivo de acesso aleatório ou randômico têm sua ordem de inserção definida por um campo denominado **chave**. O campo chave de um registro não pode ter o seu conteúdo repetido; ele deve ser único no arquivo para que possibilite a identificação do registro. Por meio do campo chave, o registro pode ser localizado diretamente, sem que haja necessidade de se percorrerem todos os registros anteriores. O campo chave deve ser declarado juntamente com as demais variáveis que irão compor o registro.

**NOTA:**

*Os arquivos de acesso aleatório ou randômico são também chamados de arquivo de acesso direto.*

Um arquivo de acesso aleatório ou randômico pode ser comparado a um CD-ROM com músicas pelo qual se pode ouvir a primeira música, depois saltar para a décima e voltar para a sétima, sem que haja necessidade de percorrer todas as músicas intermediárias.

Para que seja possível a manipulação de arquivos de acesso aleatório ou randômico, será necessário:

- Declarar o registro e o arquivo.
- Declarar as variáveis de arquivo e registro.
- Abrir o arquivo.
- Fechar o arquivo.

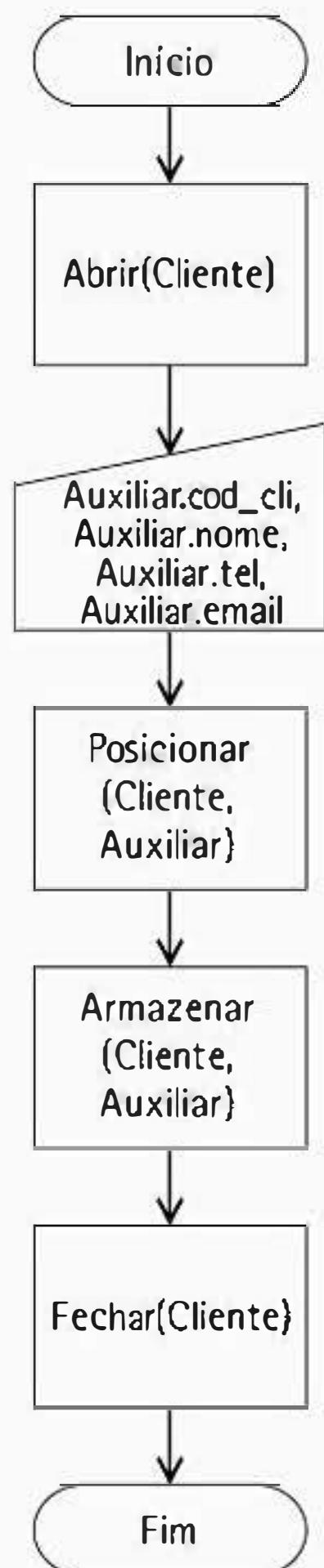
Esses passos são utilizados para qualquer operação de manipulação de arquivos e já foram explicados anteriormente no Algoritmo 9.1 — operação de inclusão.

No exemplo a seguir, será feita a construção de um cadastro de clientes que armazene código do cliente, nome, telefone e e-mail.

**EXEMPLO 9.5:** Operação de inclusão — arquivo de acesso direto.

```
1. Algoritmo Exemplo_9.5
2. Var tipo reg_cliente = registro
3. Cod_cli: inteiro
4. Nome: caracter
5. Tel: caracter
6. Email: caracter
7. Fim_registro
8. Tipo arq_cliente: arquivo direto de cliente
9. Auxiliar: reg_cliente
10. Cliente: arq_cliente
11. Inicio
12. Abrir (Cliente)
13. Ler(auxiliar.cod_cli, auxiliar.nome, auxiliar.tel,
auxiliar.email)
14. Posicionar(Cliente, Auxiliar)
15. Armazenar(Cliente, Auxiliar)
16. Fechar (Cliente)
17. Fim.
```

A instrução **Posicionar**(*nome da variável de arquivo, nome da variável de registro*) é utilizada para posicionar o novo registro corretamente. Essa posição é determinada pelo campo chave. No Algoritmo 9.5, o campo chave é o **Cod\_Cli**. É necessário que seja indicado também o armazenamento do registro para que este seja definitivamente inserido no arquivo por meio da instrução **Armazenar**(*nome da variável de arquivo, nome da variável de registro*).

**Fluxograma:****Java:**

```
1. public class regAgenda{
2. private int cod_cli;
3. private String nome;
4. private String tel;
5. private String email;
6.
7. public regAgenda (int cod, String nom, String telef, String
e_mail){
8. cod_cli = cod;
9. nome = nom;
10. tel = telef;
11. email = e_mail;
12. }
13. public int mostraCod (){
14. return cod_cli;
15. }
```

```

16. public String mostraNome (){
17. return nome;
18. }
19. public String mostraTel (){
20. return tel;
21. }
22. public String mostraEmail (){
23. return email;
24. }
25. }
```

A classe `regAgenda` foi modificada para esse algoritmo com alteração nos atributos e inclusão dos métodos para retornar seus valores, seguindo a mesma idéia do acesso a arquivos seqüenciais tratado anteriormente.

```

1. import java.io.*;
2. class Exemplo95{
3. static StringBuffer memoria = new StringBuffer();
4. public static void main(String[] args) {
5. try{
6. BufferedReader entrada;
7. entrada = new BufferedReader (new InputStreamReader
8. (System.in));
9. BufferedReader argentrada;
10. argentrada = new BufferedReader (new FileReader
11. ("Cliente.txt"));
12. String linha = "";
13. while ((linha = argentrada.readLine ()) != null) {
14. memoria.append (linha + "\n");
15. }
16. int Cod_cli = 1;
17. if (memoria.length () != 0){
18. int inicio = memoria.lastIndexOf ("#");
19. int fim = memoria.indexOf ("\t", inicio);
20. Cod_cli = Integer.parseInt (ler (inicio + 1, fim));
21. Cod_cli += 1;
22. }
23. System.out.println ("Codigo do Cliente: " + Cod_cli);
24. System.out.println ("Digite o nome");
25. String Nome = entrada.readLine();
26. System.out.println ("Digite o telefone");
27. String Tel = entrada.readLine();
28. System.out.println ("Digite o email");
29. String Email = entrada.readLine();
30. regAgenda regAg1 = new regAgenda (Cod_cli, Nome, Tel,
31. Email);
32. linha = "#" + regAg1.mostraCod () + "\t" +
```

```
30. regAg1.mostraNome () + "\t" +
31. regAg1.mostraTel () + "\t" +
32. regAg1.mostraEmail () + "\n";
33. gravar (linha);
34. } catch (FileNotFoundException erro){
35. System.out.println ("Arquivo nao encontrado!");
36. } catch (Exception erro){
37. System.out.println ("Erro de Leitura!");
38. }
39. }
40. public static String ler (int primeiro, int ultimo){
41. String dados = "";
42. char [] destino = new char [ultimo - primeiro];
43. memoria.getChars (primeiro, ultimo, destino, 0);
44. for (int i = 0; i < destino.length; i++) {
45. dados += destino [i];
46. }
47. return dados;
48. }
49. public static void gravar (String Linha){
50. try{
51. BufferedWriter saida;
52. saida = new BufferedWriter (new FileWriter
("Cliente.txt", true));
53. saida.write (Linha);
54. saida.flush ();
55. saida.close ();
56. } catch (Exception erro){
57. System.out.println ("Erro de gravacao !");
58. }
59. }
60. }
```

Essa implementação do algoritmo considera que os códigos dos clientes são numerados a partir de 1. O programa efetua uma busca no contêiner `memoria`, desde que ele não esteja vazio (verificação feita na linha 15), para obter o código atribuído ao último cliente cadastrado. Como o arquivo de dados é um arquivo texto, utilizou-se um marcador (caractere #) que precede todos os números de código dos clientes. Dessa forma, a posição do último # é obtida com a busca feita na linha 16 e o fim do campo do código é obtido com a instrução da linha seguinte. Passamos as posições dos índices de início e fim para o método `ler`, que retorna o código. Como esse código está representado por caracteres, já que se origina de um arquivo texto, o resultado obtido é uma string de caracteres. Utilizamos o método `parseInt` da classe `Integer` para converter a string para inteiro. Como esse código é o que foi atribuído ao último cliente cadastrado, ele precisa ser acrescido de 1, garantindo a seqüência e a unicidade de identifica-

ção. Obtêm-se os demais dados por meio da entrada via teclado e instancia-se um objeto, como feito anteriormente.

Para gravar os dados no arquivo, utilizaremos uma variável `linha`, do tipo `String`, que será passada como parâmetro para o método `gravar`. A variável `linha` recebe a concatenação dos diversos campos do objeto, retornados pelos respectivos métodos e dos caracteres separadores, inclusive o caractere `#` indicador do número de código do cliente (linha 29).

**ATENÇÃO!**

*Lembre-se de que o arquivo precisa ser criado antecipadamente, com o mesmo nome usado no código.*

**EXEMPLO 9.6:** Operação de consulta — arquivo de acesso aleatório.

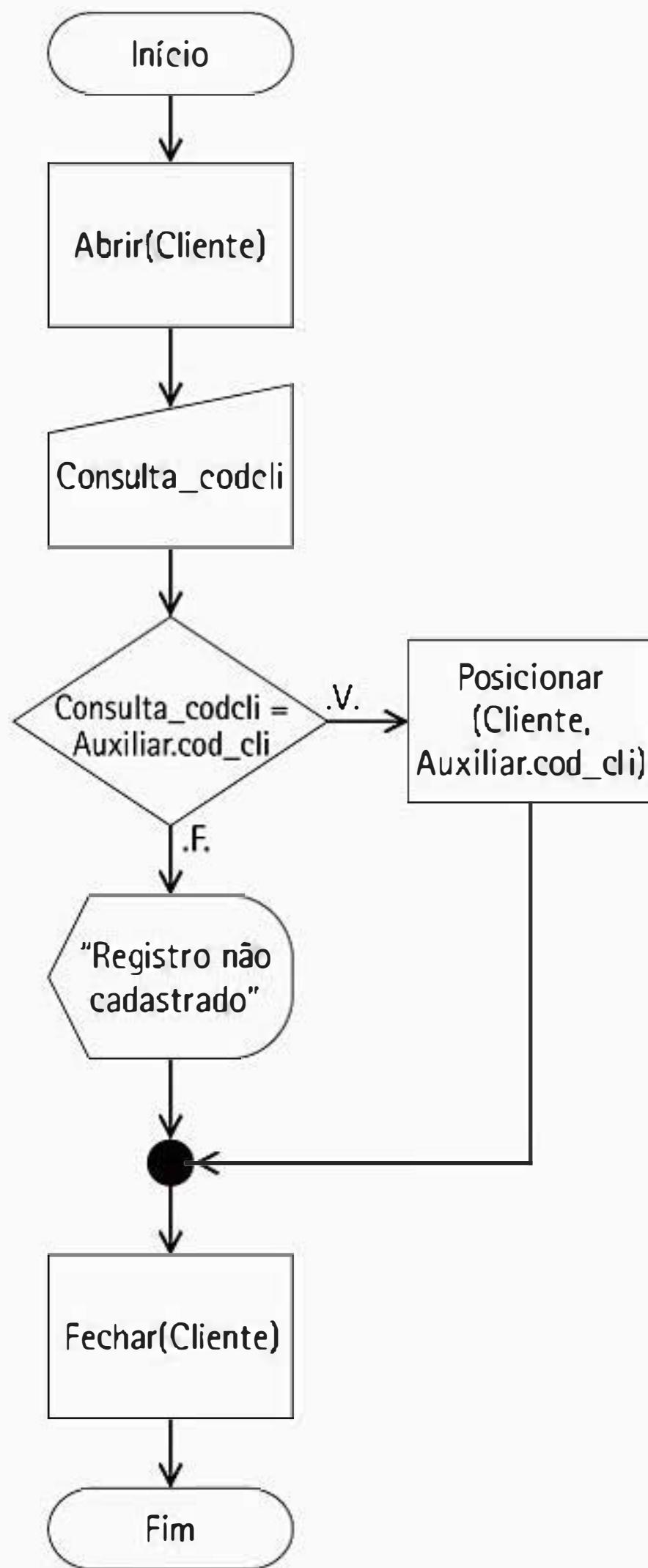
```

1. Algoritmo Exemplo_9.6
2. Var tipo reg_cliente = registro
3. Cod_cli: inteiro
4. Nome: caracter
5. Tel: caracter
6. Email: caracter
7. Fim_registro
8. Tipo arq_cliente: arquivo direto de cliente
9. Auxiliar: reg_cliente
10. Cliente: arq_cliente
11. Consulta_codcli: inteiro
12. Inicio
13. Abrir (Cliente)
14. Ler(Consulta_codcli)
15. Se (Consulta_codcli = Auxiliar.Cod_cli) Então
16. Posicionar(Cliente, Auxiliar.Cod_codcli)
17. Senão
18. Mostrar("Registro não cadastrado")
19. Fim-Se
20. Fechar (Cliente)
21. Fim.

```

**NOTA:**

● acesso direto/aleatório ao registro a ser consultado é feito por meio do campo chave.

**Fluxograma:****Java:**

```

1. import java.io.*;
2. class Exemplo96{
3. static StringBuffer memoria = new StringBuffer();
4. public static void main(String[] args){
5. try{
6. BufferedReader entrada;
7. entrada = new BufferedReader (new InputStreamReader
8. (System.in));
9. BufferedReader argentrada;
10. argentrada = new BufferedReader (new FileReader
11. ("Cliente.txt"));
12. String linha = "";
13. while ((linha = argentrada.readLine ()) != null) {
14. memoria.append(linha);
15. }
16. System.out.println(memoria);
17. } catch (IOException e) {
18. e.printStackTrace();
19. }
20. }
21. }

```

```

12. memoria.append (linha + "\n");
13. }
14. System.out.println ("Digite o codigo:");
15. String consultaCod_cli = entrada.readLine ();
16. int inicio = -1;
17. if (memoria.length () != 0){
18. inicio = memoria.indexOf ("#" + consultaCod_cli);
19. if (inicio == -1){
20. System.out.println ("Codigo nao encontrado.");
21. }
22. }else{
23. System.out.println ("O arquivo esta vazio.");
24. }
25. } catch (FileNotFoundException erro){
26. System.out.println ("Arquivo nao encontrado!");
27. } catch (Exception erro){
28. System.out.println ("Erro de Leitura!");
29. }
30. }
31. }
```

A implementação da busca direta é feita por meio do código do cliente, resultante da entrada via teclado, e do caractere associado a esse código (caractere #). A instrução na linha 18 executa essa operação e o método usado é o mesmo dos anteriores: `indexOf`, que retoma a posição inicial do dado pesquisado no contêiner de memória e, consequentemente, do arquivo originário.

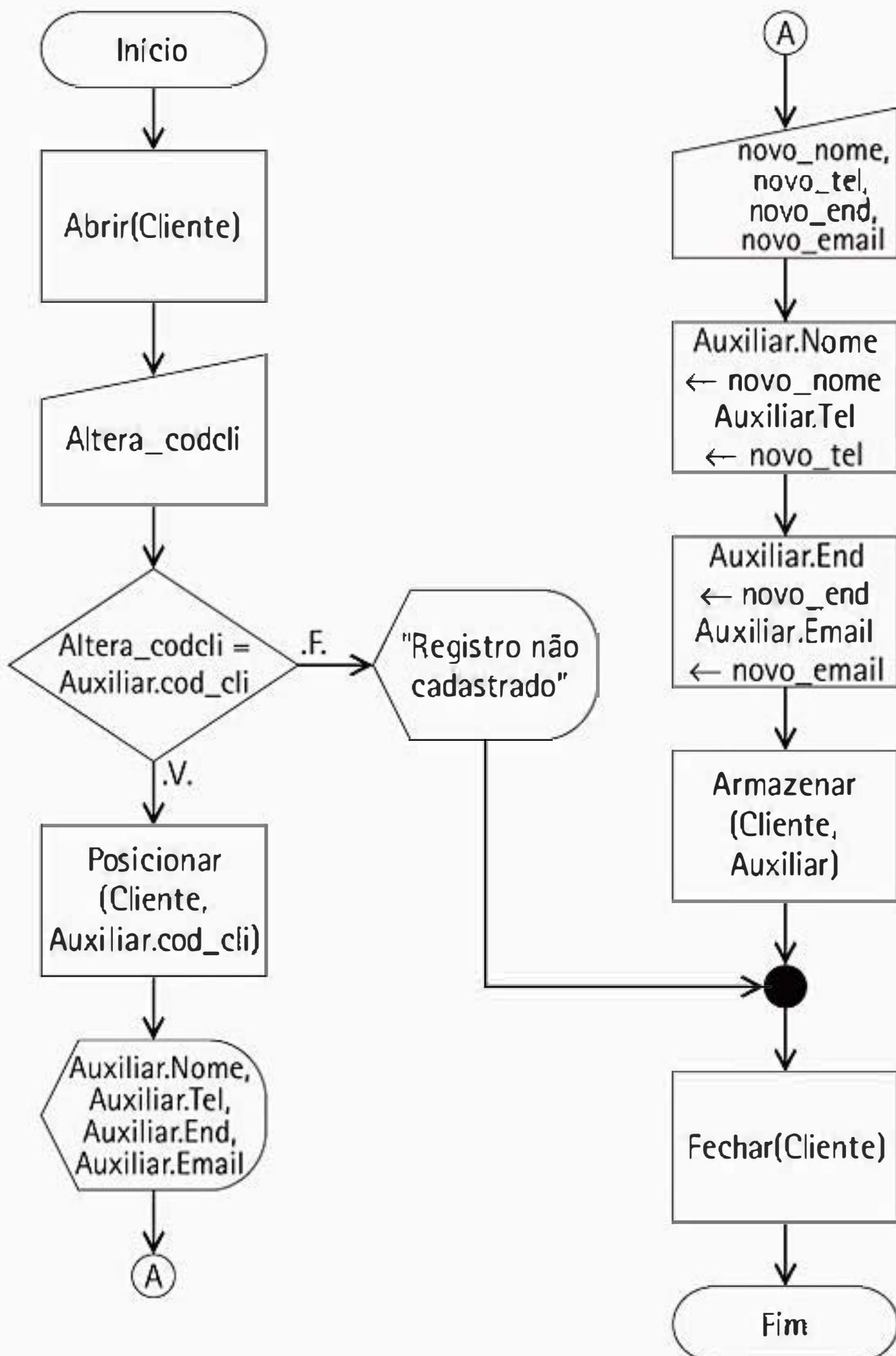
Foram introduzidos dois testes de verificação para garantir a execução correta do programa. Um deles, na linha 17, confere se o contêiner de memória está vazio, executando a busca somente no caso de existirem dados, ou seja, o tamanho de `memoria` obtido pelo método `length` deve retomar um valor diferente de 0. A outra verificação indica se a variável `inicio` recebeu um valor diferente do atribuído quando de sua inicialização (-1), significando que o código foi encontrado. Caso o código tenha sido encontrado, a variável `inicio` terá um valor igual ou maior que 0 e corresponderá à posição de início do campo pesquisado.

#### **EXEMPLO 9.7:** Operação de alteração — arquivo de acesso aleatório.

```

1. Algoritmo Exemplo_9.7
2. Var Tipo reg_cliente = registro
3. Cod_cli: inteiro
4. Nome: caracter
5. Tel: caracter
6. Email: caracter
7. Fim_registro
8. Tipo arq_cliente : arquivo direto de cliente
```

```
9. Auxiliar : reg_cliente
10. Cliente : arq_cliente
11. Altera_codcli: inteiro
12. novo_nome, novo_tel, novo_end, novo_email : caracter
13. Inicio
14. Abrir (Cliente)
15. Ler(Altera_codcli)
16. Se (Altera_codcli = Auxiliar.Cod_cli) Então
17. Posicionar(Cliente, Auxiliar.Cod_codcli)
18. Mostrar(Auxiliar.Nome, Auxiliar.Tel, Auxiliar.End,
19. Auxiliar.Email)
20. Ler(novo_nome, novo_tel, novo_end, novo_email)
21. Auxiliar.Nome ← novo_nome
22. Auxiliar.Tel ← novo_tel
23. Auxiliar.End ← novo_end
24. Auxiliar.Email ← novo_email
25. Armazenar(Cliente, Auxiliar)
26. Senão
27. Mostrar("Registro não cadastrado")
28. Fim-Se
29. Fechar (Cliente)
29. Fim.
```

**Fluxograma:****Java:**

A classe `regAgenda` deve ser modificada para a inclusão dos métodos que permitem alteração dos atributos `nome`, `tel` e `email`; tais métodos devem ser compatíveis com os nomes utilizados no programa desse exemplo: `alteraNome`, `alteraTel` e `alteraEmail`. Caso você queira atribuir outros nomes aos métodos, não há problema, desde que eles sejam compatíveis (na classe `regAgenda` e no Exemplo97), lembrando a questão das letras maiúsculas e minúsculas, que não pode ser desprezada, pois Java é *case-sensitive*.

```

1. import java.io.*;
2. class Exemplo97{
3. static StringBuffer memoria = new StringBuffer();
4. public static void main(String[] args){
5. try{

```

```
6. BufferedReader entrada;
7. entrada = new BufferedReader (new InputStreamReader
8. (System.in));
9. BufferedReader arqentrada;
10. arqentrada = new BufferedReader (new FileReader
11. ("Cliente.txt"));
12. String linha = "";
13. while ((linha = arqentrada.readLine ()) != null) {
14. memoria.append (linha + "\n");
15. }
16. System.out.println ("Digite o codigo:");
17. String alteraCod_cli = entrada.readLine ();
18. int primeiro = -1;
19. if (memoria.length () != 0){
20. primeiro = memoria.indexOf ("#" + alteraCod_cli);
21. if (primeiro != -1){
22. int fim = memoria.indexOf ("\t", primeiro);
23. int Cod_cli = Integer.parseInt (ler (primeiro += 1,
24. fim));
25. primeiro = fim + 1;
26. fim = memoria.indexOf ("\t", primeiro);
27. String Nome = ler (primeiro, fim);
28. int inicio = fim + 1;
29. fim = memoria.indexOf ("\t", inicio);
30. String Tel = ler (inicio, fim);
31. inicio = fim + 1;
32. fim = memoria.indexOf ("\n", inicio);
33. String Email = ler (inicio, fim);
34. System.out.println (Nome + " " + Tel + " " +
35. Email);
36. regAgenda regAg1 = new regAgenda (Cod_cli, Nome,
37. Tel, Email);
38. System.out.println ("Entre com novo nome:");
39. Nome = entrada.readLine ();
40. System.out.println ("Entre com novo telefone:");
41. Tel = entrada.readLine ();
42. System.out.println ("Entre com novo email:");
43. Email = entrada.readLine ();
44. regAg1.alteraNome (Nome);
45. regAg1.alteraTel (Tel);
46. regAg1.alteraEmail (Email);
47. memoria.replace (primeiro, fim, regAg1.mostraNome
48. () + "\t" + regAg1.mostraTel () + "\t" + regAg1.mostraEmail ());
49. gravar ();
50. }else{
51. System.out.println ("Codigo nao encontrado.");
52. }
53. }else{
```

```

48. System.out.println ("O arquivo esta vazio.");
49. }
50. arqentrada.close ();
51. } catch (FileNotFoundException erro){
52. System.out.println ("Arquivo nao encontrado!");
53. } catch (Exception erro){
54. System.out.println ("Erro de leitura!");
55. }
56. }
57. public static String ler (int primeiro, int ultimo){
58. String dados = "";
59. char [] destino = new char [ultimo - primeiro];
60. memoria.getChars (primeiro, ultimo, destino, 0);
61. for (int i = 0; i < destino.length; i++){
62. dados += destino [i];
63. }
64. return dados;
65. }
66. public static void gravar (){
67. try{
68. BufferedWriter saida;
69. saida = new BufferedWriter (new FileWriter
("Cliente.txt"));
70. saida.write (memoria.toString ());
71. saida.flush ();
72. saida.close ();
73. } catch (Exception erro){
74. System.out.println ("Erro de gravacao!");
75. }
76. }
77. }

```

O código do cliente a ser pesquisado é atribuído à variável `alteraCod_cli` e a busca é feita nos moldes do Exemplo 9.6. Porém, neste caso, utiliza-se uma variável adicional: `primeiro`, que armazena a posição inicial do trecho que será alterado quando da atualização do contêiner `memoria`. A variável `primeiro` é utilizada durante o percurso pelo registro até que ela represente a posição inicial do campo `Nome`, quando então se passa a utilizar a variável `inicio` em conjunto com a variável `fim` para o percurso e a recuperação dos demais campos do registro. Isso se deve ao fato de que se pretende preservar o campo código do cliente, que é um campo de controle e não pode ser alterado pelo usuário.

Na linha 18, a variável `primeiro` recebe a posição inicial do caractere `#` e o código do cliente, caso ele tenha sido encontrado. Na linha 19, testa-se a condição de `primeiro` ter recebido um índice válido. Na linha 20, efetua-se a busca a partir da posição encontrada e a próxima posição de tabulação, isolando-se as posições de início e fim do código do cliente. Observe que, para a passagem de posições para o método

ler, a variável `primeiro` é incrementada em 1, visto que o campo propriamente dito inicia-se a partir da posição seguinte, que corresponde aos caracteres do número do código do cliente. Deve-se desprezar o caractere `#`. Na linha 21, a variável `Cod_cli` recebe os dados do código do cliente, que é convertido para inteiro, seu tipo de dado original, como foi declarado em `regAgenda`. Na linha 22, a variável `primeiro` recebe a posição `fim + 1`, pois o próximo campo a ser recuperado é o campo `Nome`, que está na seqüência. Sua primeira posição equivale a uma posição além do final do campo código. Na linha 25 passa-se a utilizar a variável `inicio`, permitindo-se que o valor da variável `primeiro` permaneça com o valor da posição que lhe foi atribuída por último. A tabela a seguir apresenta resumidamente um exemplo de possíveis posições que os campos de um registro poderiam assumir e as variáveis utilizadas para recuperar as posições de cada campo. Logo a seguir mostramos um teste de mesa aplicado a esse exemplo. A coluna ‘LINHA’ corresponde à linha do programa.

É importante destacar que, no momento em que é feita a atualização dos dados no contêiner da memória, são passadas as posições referentes ao valor das variáveis `primeiro` e `fim`, que correspondem ao intervalo de caracteres que vai de `Nome` até `Email`. No exemplo, esse intervalo vai da posição 3 até a 39.

| posição  | 0        | 1        | 2   | 3 a 8    | 9   | 10 a 17 | 18  | 19 a 38 | 39  |
|----------|----------|----------|-----|----------|-----|---------|-----|---------|-----|
| conteúdo | #        | Código   | \t  | Nome     | \t  | Tel     | \t  | Email   | \n  |
| variável | primeiro | primeiro | fim | primeiro | fim | inicio  | fim | inicio  | fim |

**TABELA 9.1 |** Posições possíveis para os campos de um registro

| LINHA | primeiro | inicio | fim |
|-------|----------|--------|-----|
| 16    | -1       |        |     |
| 18    | 0        |        |     |
| 20    | 0        |        | 2   |
| 22    | 3        |        | 2   |
| 23    | 3        |        | 9   |
| 25    | 3        | 10     | 9   |
| 26    | 3        | 10     | 18  |
| 28    | 3        | 19     | 18  |
| 29    | 3        | 19     | 39  |

|TABELA 9.2| Teste de mesa

**Exemplo 9.8:** Operação de exclusão — arquivo de acesso direto.

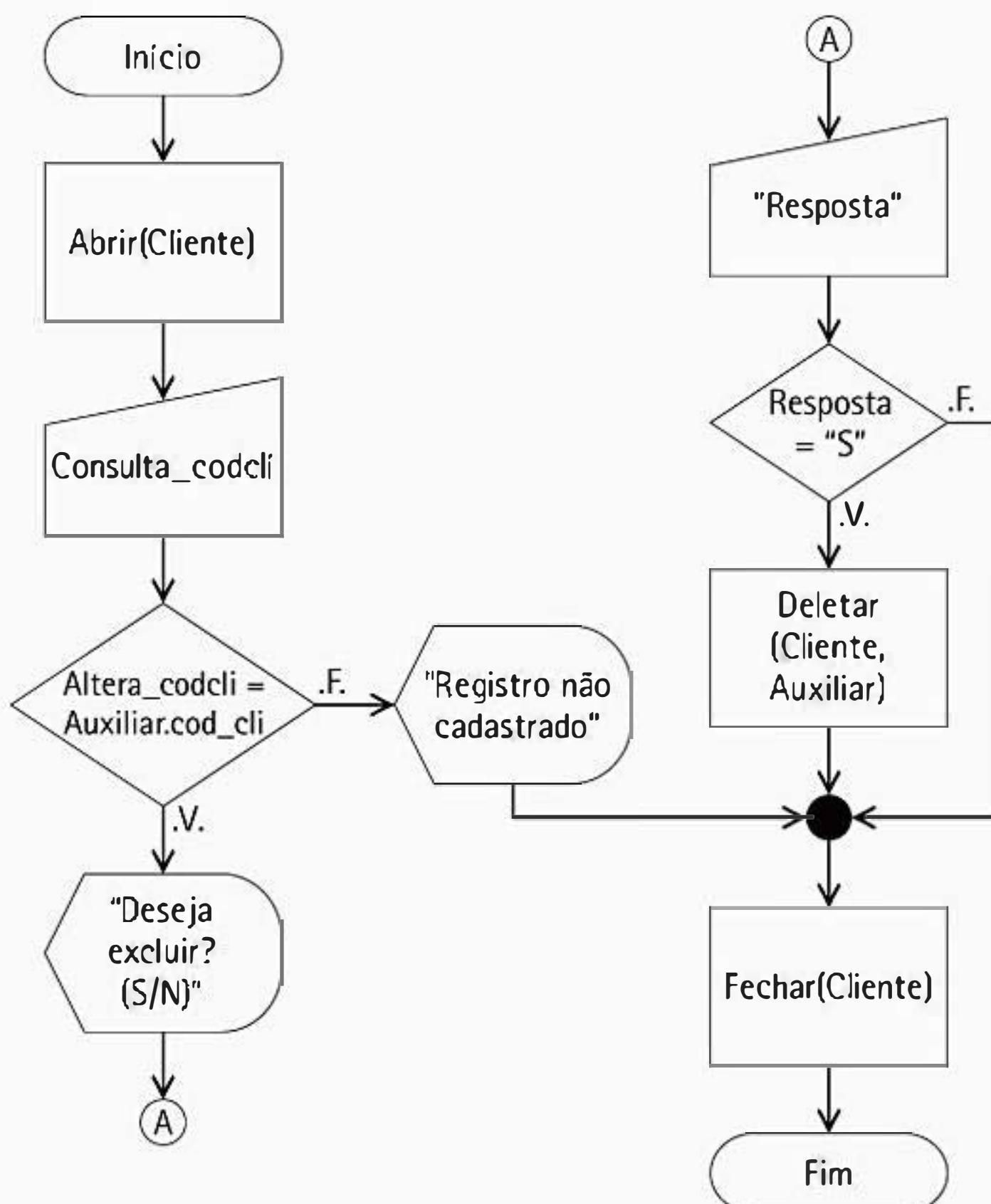
- ```
1. Algoritmo Exemplo_9.8
2. Var tipo reg_cliente = registro
3.           Cod_cli: inteiro
4.           Nome: caracter
5.           Tel: caracter
```

```

6.                               Email: caracter
7.                               Fim_registro
8.   Tipo arq_cliente: arquivo direto de cliente
9.   Auxiliar: reg_cliente
10.  Cliente: arq_cliente
11.  Consulta_codcli: inteiro
12.  Resposta: caracter
13. Inicio
14.  Abrir (Cliente)
15.  Ler(Consulta_codcli)
16.  Se (Consulta_codcli = Auxiliar.Cod_cli) Então
17.    Posicionar(Cliente, Auxiliar.Cod_codcli)
18.    Mostrar("Deseja Excluir? S/N") Ler(Resposta)
19.    Se (Resposta = "S") Então
20.      Deletar(Cliente, Auxiliar)
21.    Fim-Se
22.  Senão
23.    Mostrar("Registro não cadastrado")
24.  Fim-Se
25.  Fechar (Cliente)
26. Fim.

```

Fluxograma:



Java:

```
1. import java.io.*;
2. class Exemplo98{
3.     static StringBuffer memoria = new StringBuffer();
4.     public static void main(String[] args){
5.         try{
6.             BufferedReader entrada;
7.             entrada = new BufferedReader (new InputStreamReader
8.             System.in));
9.             BufferedReader argentrada;
10.            argentrada = new BufferedReader (new FileReader
11.            ("Cliente.txt"));
12.            String linha = "";
13.            while ((linha = argentrada.readLine ()) != null) {
14.                memoria.append (linha + "\n");
15.            }
16.            System.out.println ("Digite o codigo:");
17.            String consultaCod_cli = entrada.readLine ();
18.            int inicio = -1;
19.            if (memoria.length () != 0){
20.                inicio = memoria.indexOf ("#" + consultaCod_cli);
21.                if (inicio != -1){
22.                    int fim = memoria.indexOf ("\t", inicio);
23.                    int Cod_cli = Integer.parseInt (ler (inicio + 1,
24.                    fim));
25.                    System.out.println ("Deseja excluir cliente: " +
26.                        Cod_cli + "? S/N");
27.                    String resposta = entrada.readLine ();
28.                    if (resposta.equalsIgnoreCase ("S")){
29.                        fim = memoria.indexOf ("\n", inicio);
30.                        memoria.delete (inicio, fim + 1);
31.                        gravar ();
32.                    }
33.                }else{
34.                    System.out.println ("Codigo nao encontrado.");
35.                }
36.                argentrada.close ();
37.            } catch (FileNotFoundException erro){
38.                System.out.println ("Arquivo nao encontrado!");
39.            } catch (Exception erro){
40.                System.out.println ("Erro de leitura!");
41.            }
42.        }
43.        public static String ler (int primeiro, int ultimo){
44.            String dados = "";
```

```

45.     char [] destino = new char [ultimo - primeiro];
46.     memoria.getChars (primeiro, ultimo, destino, 0);
47.     for (int i = 0; i < destino.length; i++){
48.         dados += destino [i];
49.     }
50.     return dados;
51. }
52. public static void gravar (){
53.     try{
54.         BufferedWriter saida;
55.         saida = new BufferedWriter (new FileWriter
("Cliente.txt"));
56.         saida.write (memoria.toString ());
57.         saida.flush ();
58.         saida.close ();
59.     } catch (Exception erro){
60.         System.out.println ("Erro de gravacao!");
61.     }
62. }
63. }
```

Nesse exemplo, fazemos a busca do código do cliente como no exemplo anterior. Obtemos, assim, o início do registro a ser excluído. Obtemos o código pelo método `ler` (linha 21), para exibi-lo no pedido de confirmação ao usuário (linha 22). Se a resposta for igual a "S" (desprezando-se maiúsculas e minúsculas — linha 25), procede-se à exclusão do registro no contêiner de memória. O método `gravar` efetiva a transferência dos dados para o arquivo.

9.5 EXERCÍCIOS PARA FIXAÇÃO

Para os exercícios abaixo, escreva o pseudocódigo, faça o fluxograma e codifique o programa em Java.

1. Elabore um cadastro que armazene os dados dos atletas de um clube. Deverão ser armazenados: nome, idade, altura, sexo, peso e modalidade esportiva. O programa deverá permitir a manipulação dos dados armazenados. Utilize o método seqüencial.
2. Elabore um programa que possibilite o armazenamento dos dados dos funcionários de uma empresa para que seja gerada a folha de pagamento. Deverão ser armazenados: nome do funcionário, código funcional, data de admissão, salário bruto, número de dependentes e cargo. O programa deverá permitir a manipulação das informações e possuir uma opção para calcular o salário líquido de cada funcionário. Utilize o método randômico.

9.6 EXERCÍCIOS COMPLEMENTARES

Para os exercícios abaixo, escreva o pseudocódigo e o fluxograma e codifique o programa em Java.

1. Elabore um controle acadêmico que permita:
 - a) o cadastro dos dados pessoais dos alunos;
 - b) o cadastro da grade escolar do aluno;
 - c) o cadastro das notas associadas à grade do aluno;
 - d) a manipulação dos dados com controle de senha.

Crie um menu de opções e utilize todos os recursos aprendidos até agora. Utilize o método randômico. O programa deverá verificar se o aluno foi aprovado ou reprovado nas disciplinas, levando-se em consideração que a média de aprovação é de 7,0 sem exame e de 5,0 com exame.

2. Elabore um controle de estoque que permita:
 - a) cadastrar novos produtos;
 - b) manipular as informações cadastradas;
 - c) acompanhar a quantidade de produtos disponíveis;
 - d) consultar o nome do produto, a quantidade disponível e o preço.

Faça um menu de opções e utilize todos os recursos aprendidos. Utilize o método seqüencial.

10

ESTRUTURAS DE
DADOS DINÂMICAS

- ▶ *Listas*
- ▶ *Filas*
- ▶ *Pilhas*
- ▶ *Árvores*

OBJETIVOS:

Apresentar as estruturas de dados freqüentemente utilizadas na programação, por meio de diagramas e exemplos simples, para facilitar o entendimento e sua implementação.

As estruturas de dados são, muitas vezes, a pedra no sapato do programador inexperiente, por isso, tentamos demonstrar de maneira bem simples como construir alguns exemplos. Como existe a possibilidade de implementação dessas estruturas com o uso de arranjos ou alocação dinâmica, para não sermos repetitivos demonstramos a implementação de listas e árvores utilizando alocação dinâmica e filas e pilhas por arranjos.

Quando falamos em listas, filas, pilhas e árvores podemos dizer que todas são na verdade ‘listas de informações’, cuja diferença principal está no acesso a essas ‘listas’ para inclusão e remoção de informações.

Os arranjos (vetores ou matrizes) são mais simples de implementar: o conteúdo da ‘lista’ é armazenado em um espaço de memória com tamanho para N elementos que serão dispostos em posições contínuas, isso é, um seguido do outro. Mas, apesar de ser mais fácil de compreender como manipular seus dados, os arranjos possuem limitação

quanto à quantidade de elementos que o conjunto irá suportar, isto é, o arranjo possui um tamanho pré-determinado que pode ou não ser totalmente ocupado, além de que pode haver a necessidade de mais espaço do que aquele que foi inicialmente reservado.

Já quando falamos em alocação dinâmica utilizamos posições descontinuadas da memória RAM; isso é possível pois cada um dos elementos da ‘lista’ deve possuir uma referência para os elementos seguinte e anterior. Essa referência é o endereço da posição de memória em que se encontra tal elemento.

A seguir serão apresentados os conceitos e exemplos de listas, listas encadeadas, filas, pilhas e árvores. As estruturas não estão apresentadas em ordem de importância e/ou facilidade para compreensão. O conceito de lista é abordado inicialmente e por uma questão de continuidade a construção de listas encadeadas é abordada em seguida.

NOTA:

Cabe lembrar que a linguagem de programação Java não suporta ponteiros, assim, quando do uso de ponteiros nos exemplos, na linguagem serão utilizados objetos que farão referência aos elementos.

10.1 LISTAS

Uma lista é uma coleção de elementos do mesmo tipo dispostos linearmente que podem ou não seguir determinada organização, por exemplo: $[E_1, E_2, E_3, E_4, E_5, \dots, E_n]$, onde n deve ser $>= 0$.

Como exemplos de listas podemos citar: lista de chamada de alunos, lista de compras de supermercado, lista telefônica, entre outros. O exemplo apresentado a seguir é de uma lista de pagamentos a serem efetuados em um mês, exemplo que também será utilizado nos tópicos seguintes.

Lista de pagamentos
Prestação do carro
Cartão de crédito
Conta de luz
Condomínio
TV a cabo
Crediário das Casas Bahia

Lista simples

Quando criamos uma lista para utilizá-la como estrutura de dados, podemos utilizar como contêiner para armazenamento dos dados um vetor ou uma matriz, então dizemos que esta é uma lista implementada por meio de arranjo; ou então podemos utilizar a alocação dinâmica, isto é, não criamos um contêiner para armazenar os dados, mas precisamos referenciar os elementos seguinte e anterior de cada elemento, então teremos uma lista encadeada.

Veja um exemplo de lista simples:

`Lista de pagamentos ← [prestação do carro, cartão de crédito, conta de luz, condomínio, TV a cabo, crediário das Casas Bahia]`

Essa é uma lista que possui seis elementos do tipo caracter, e os elementos estão armazenados em um vetor.

10.1.1 LISTAS ENCADEADAS

Uma lista encadeada é um conjunto de elementos que estão dispostos em uma dada organização física não linear, isto é, estão espalhados pela memória. Para organizar a lista de maneira que possa ser utilizada como um conjunto linear, é necessário que cada elemento do conjunto possua informações sobre o seu elemento anterior e o seu elemento seguinte. Para exemplificar será utilizada uma lista de pagamentos que devem ser efetuados no mês. Os pagamentos estão dispostos em uma ordem aleatória, isto é, linear:

Lista de pagamentos
Prestação do carro
Cartão de crédito
Conta de luz
Condomínio
TV a cabo
Crediário das Casas Bahia

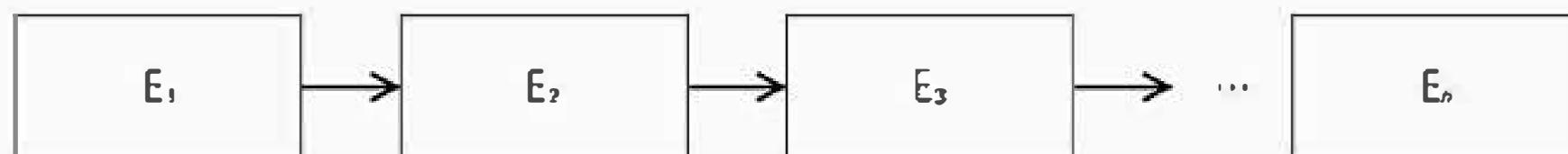
lista simples

Olhando para a lista pode-se perceber qual é o primeiro elemento, qual é o segundo elemento e assim por diante, mas quando desejamos implementar essa lista em uma estrutura de dados precisamos dizer qual será o próximo elemento. Para isso, cada elemento da lista é representado por um nó, e cada nó deve conter os dados e um campo que indique qual é o próximo elemento da lista — esse campo é chamado de ponteiro. Observe a lista seguinte:

Lista de pagamentos	Ponteiro para o próximo elemento
Prestação do carro	2
Cartão de crédito	3
Conta de luz	4
Condomínio	5
TV a cabo	6
Crediário das Casas Bahia	Obs.: este é o último elemento do conjunto, então não aponta para nenhum outro.

lista com um campo para encadeamento

O elemento 1 aponta para o elemento 2, o elemento 2 aponta para o elemento 3 e assim por diante:

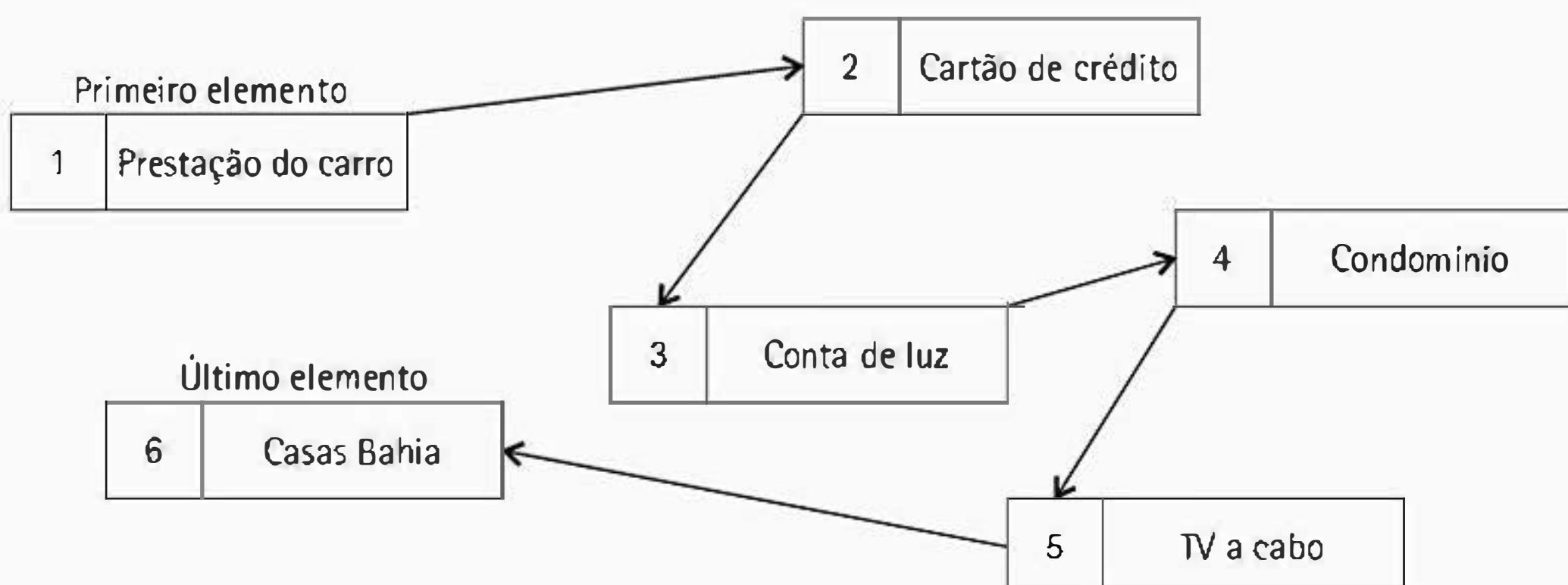


| FIGURA 10.1 | Listas encadeadas

onde:

- o primeiro elemento da lista é E_1 ;
- o último elemento da lista é E_n ;
- o predecessor de E_2 é E_1 ;
- o sucessor de E_2 é E_3 ;
- e assim por diante até o último elemento.

De acordo com o exemplo apresentado teremos:



| FIGURA 10.2 | Listas encadeadas

Trata-se de uma lista de encadeamento simples, onde:

- o primeiro elemento da lista, ou seja, o seu começo, é prestação do carro;
- o seu sucessor é cartão de crédito, que tem como predecessor prestação do carro e assim por diante;
- o último elemento da lista, ou seja, o seu final, é crediário das Casas Bahia.

NOTA: O ponteiro guarda o endereço de memória do elemento; o exemplo acima é hipotético.

Na representação algorítmica é bastante simples ilustrarmos esses apontadores ou ponteiros, mas a linguagem de programação Java não aceita ponteiros — eles são representados por uma referência ao elemento.

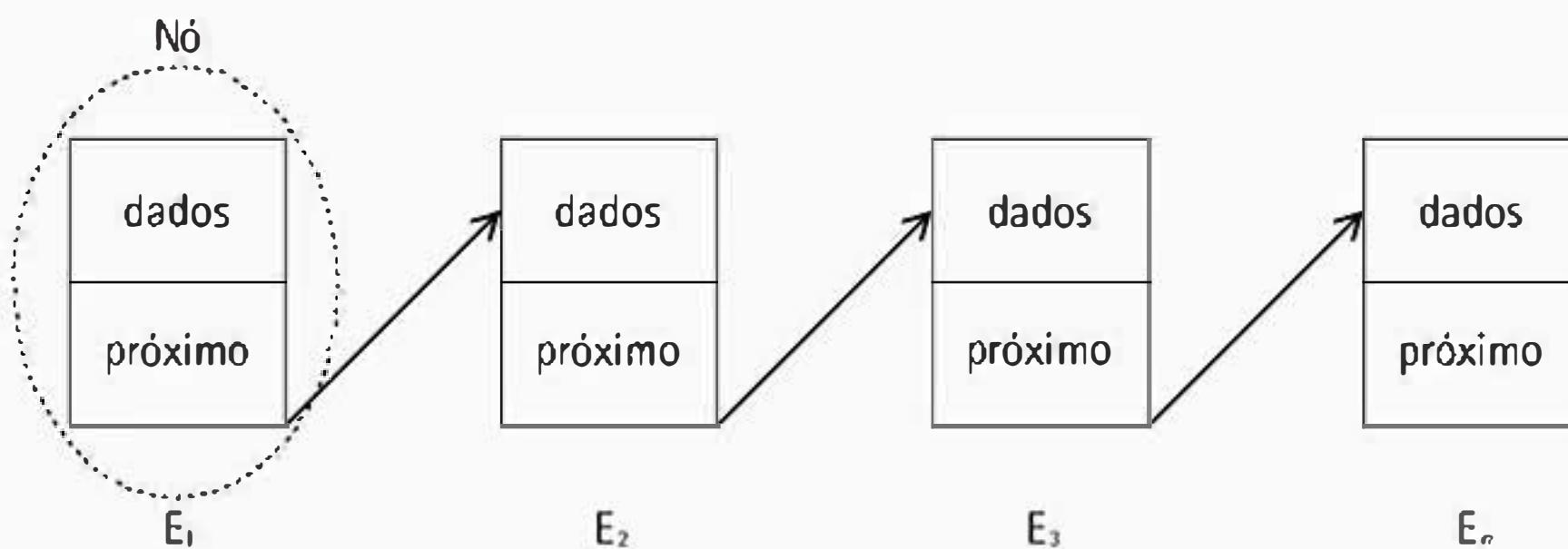
10.1.2 TIPOS DE LISTAS ENCADEADAS

As listas encadeadas podem ser do tipo:

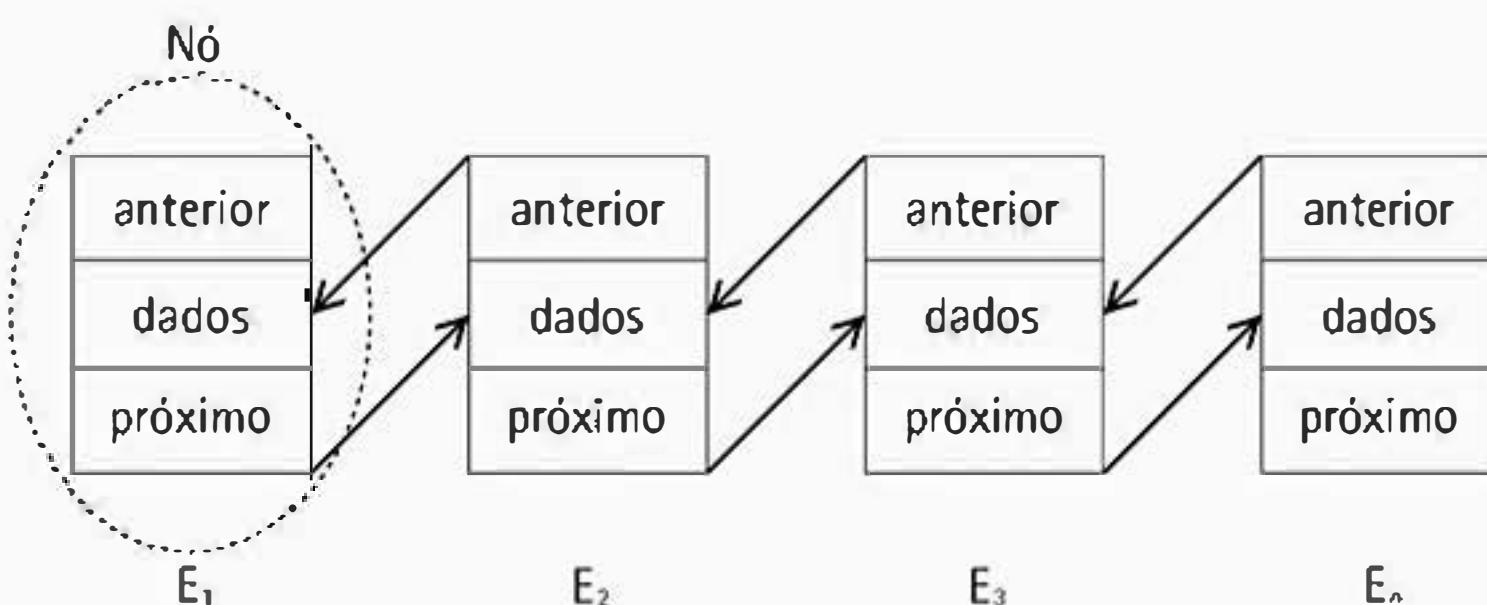
- **Encadeamento simples:** os elementos da lista possuem apenas um ponteiro que aponta para o elemento sucessor ou próximo (como no exemplo apresentado anteriormente), como mostra a Figura 10.3.
- **Duplamente encadeadas:** cada elemento possui um campo que aponta para o seu predecessor (anterior) e outro para o seu sucessor (próximo). Veja a Figura 10.4.
- **Ordenadas:** a ordem linear da lista corresponde à ordem linear dos elementos, isto é, quando um novo elemento é inserido na lista ele deve ser colocado em tal posição que garanta que a ordem da lista será mantida; essa ordem pode ser definida por um campo da área de dados, como por exemplo se tivermos uma lista ordenada com seguintes valores [1, 5, 7, 9] e desejarmos incluir um novo elemento com o valor 6, este valor deverá ser incluído entre os valores 5 e 7 (Figura 10.5.).

Uma lista ordenada pode ser de encadeamento simples ou duplo, mas o princípio para a ordenação é o mesmo.

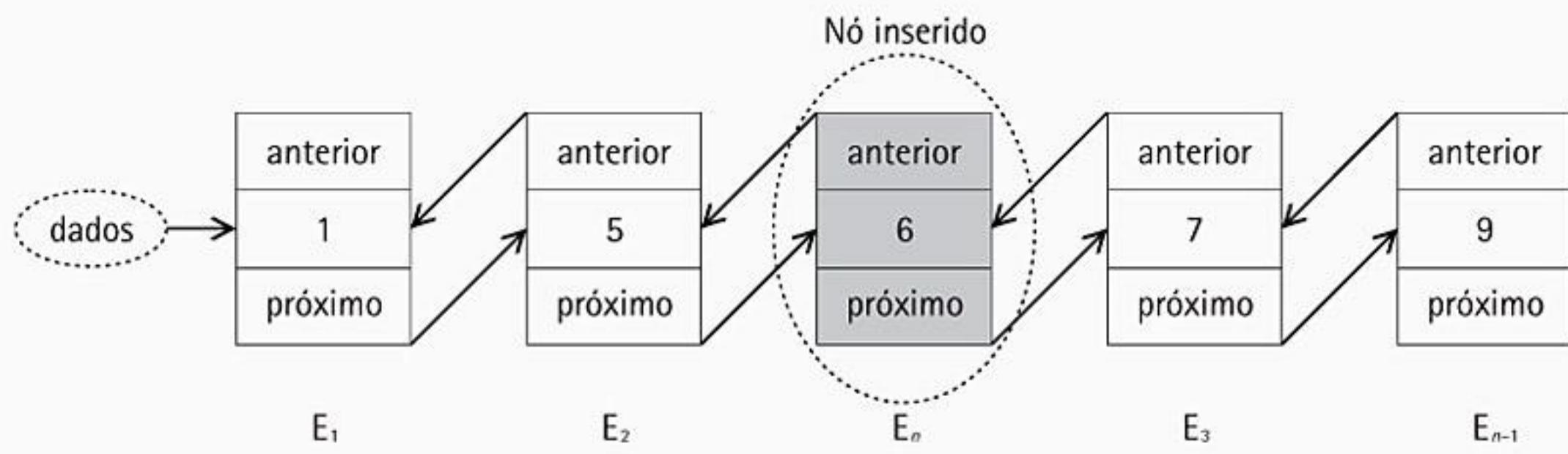
- **Circulares:** o ponteiro próximo do último elemento aponta para o primeiro; e o ponteiro anterior do primeiro elemento aponta para o último. Na Figura 10.6, E_1 é o primeiro elemento e E_n , o último elemento.



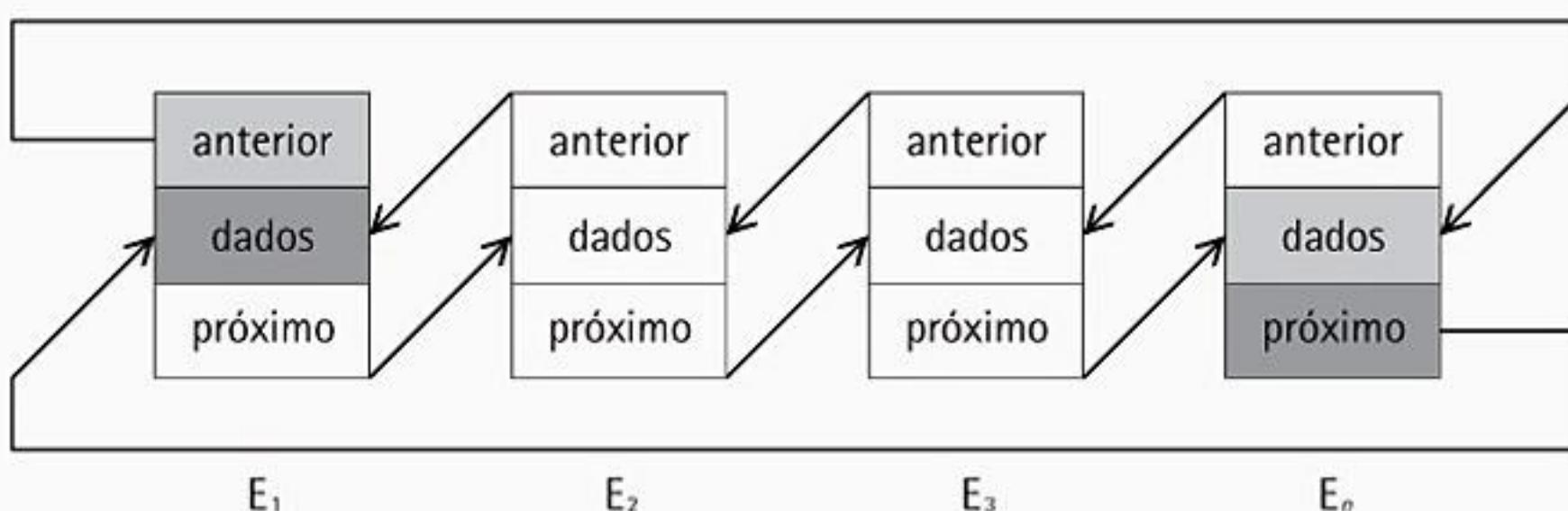
| FIGURA 10.3 | Listas com encadeamento simples



| FIGURA 10.4 | Listas duplamente encadeadas



|FIGURA 10.5| Listas ordenadas



|FIGURA 10.6| Listas circulares

Neste livro, para exemplificarmos as explicações, adotaremos as listas simples e duplamente encadeadas.

LEMBRE-SE:

As listas podem ser implementadas por meio de arranjos ou apontadores. Na implementação por meio de arranjos os elementos da lista são armazenados em posições de memória seguidas. Os dados são armazenados em uma matriz com tamanho pré-definido.

As listas implementadas por meio de apontadores permitem que os seus elementos sejam armazenados em posições descontínuas na memória, tornando mais fáceis as operações de inserção e remoção de elementos.

Essas formas de implementação são válidas também para filas, pilhas e árvores devido à similaridade na construção; como vocês poderão notar, a diferença entre essas estruturas refere-se à manipulação.

10.2 LISTAS DE ENCADEAMENTO SIMPLES

Agora que você já sabe o que é uma lista simples vamos implementá-la. A seguir será apresentado o algoritmo que representa a criação e a manipulação de uma lista de encadeamento simples.

EXEMPLO 10.1: Criação e manipulação de uma lista simples.

```

1. Algoritmo ExemploListaSimples
2. Tipo apontador: ^NoSimples
3.     NoSimples = registro
4.             valor: inteiro
5.             prox: apontador
6.         fim
7.     ListaSimples = registro
8.             primeiro: apontador
9.             ultimo: apontador
10.        fim
11. Inicio
12.     ListaSimples.primeiro ← nulo
13.     ListaSimples.ultimo ← nulo
14.     Procedimento InsereNo_fim (var novoNo: NoSimples)
15. inicio
16.     novoNo^.prox ← nulo
17.     Se {ListaSimples.primeiro = nulo} Então
18.         ListaSimples.primeiro ← novoNo
19.     Fim-Se
20.     Se (ListaSimples.ultimo <> nulo) Então
21.         ListaSimples.ultimo^.prox ← novoNo
22.     Fim-Se
23.     ListaSimples.ultimo ← novoNo
24. Fim
25.     Procedimento excluiNo{var elemento: inteiro}
26. var
27.     temp_no: NoSimples
28.     temp: NoSimples
29. inicio
30.     temp ← ListaSimples.primeiro
31.     temp_no ← ListaSimples.primeiro^.prox
32.     Enquanto (temp_no^.prox <> nulo ou temp_no.valor <>
elemento) faça
33.         temp ← temp_no
34.         temp_no ← temp_no^.prox
35.     Fim-Enquanto
36.     temp^.prox ← temp_no^.prox
37.     desposicione (temp_no)
38. Fim.

```

Na linha 2 a variável apontador é um tipo construído que terá a função de referenciar o próximo elemento de cada um dos elementos da lista, por isso ele deve ser declarado como sendo do mesmo tipo da variável que irá representar o nó. Em nosso exemplo, o nó está sendo representado pelo NoSimples. O ^ que precede o tipo de dado é utilizado para representar a função de 'apontar para' (^NoSimples), isto é, a variável apontador sempre apontará para algum outro elemento do tipo NoSimples.

Note, na linha 3, a declaração do `NoSimples`, também um tipo construído. Ele possui as variáveis `valor`, que é do tipo inteiro, e `prox`, que é do tipo apontador. Você se recorda de que o apontador irá fazer referência a um outro elemento; nesse caso ele irá fazer referência ao próximo nó do elemento em questão.

Na linha 7 está sendo declarada a `ListaSimples`, que também é um tipo construído; é um registro que contém as variáveis `primeiro` e `ultimo`, que serão utilizadas na construção da lista.

LEMBRE-SE: *Isso será possível pois o `primeiro^.prox` irá apontar para o próximo elemento, e o próximo para o próximo e assim por diante, pois a variável apontador é do tipo referência para `NoSimples`.*

O procedimento `InsereNo`, linha 14, recebe o parâmetro `novoNo`, que é uma variável do tipo `NoSimples`, então lembre-se que `novoNo` é um registro que possui as variáveis `valor` e `prox`. Esse procedimento é utilizado para inserir um novo nó no final da lista. Note que o `novoNo^.prox` recebe `nulo` — isso deve ser feito para que esse nó seja considerado o último.

Nas linhas seguintes é verificado se esse deverá ser o primeiro elemento da lista no teste `ListaSimples.primeiro = nulo`; se isso for verdadeiro, então o `novoNo` será o primeiro da lista.

Na linha 20 é verificado se o último nó é diferente de nulo, quando já existem elementos na lista. Isso é feito para que o `novoNo` seja inserido na última posição da lista.

Na linha 23 a variável `ListaSimples.ultimo` recebe o valor do `novoNo`.

DICA: *Se o `prox` for nulo, significa que esse elemento é o último.*

Se o primeiro elemento e o último tiverem valor nulo, significa que a lista está vazia.

Se o `prox` do primeiro elemento for nulo, significa que a lista só tem um elemento.

O procedimento `excluiNo`, linha 25, recebe como parâmetro um valor para a variável `elemento`, que é do tipo inteiro, e irá guardar o dado do nó que deverá ser excluído, nesse caso para a variável `valor`. Esse procedimento também utiliza as variáveis `temp` e `temp_no`, que serão variáveis auxiliares do tipo `NoSimples`. Nas linhas 30 e 31 essas variáveis recebem, respectivamente, o valor do primeiro nó e o valor do próximo nó (do primeiro).

DICA: *É aconselhável criar uma rotina para verificar se a lista está vazia ou com apenas um elemento antes de implementar esse método, senão, caso esteja, irá ocasionar um erro. Isso pode ser feito com o seguinte teste, que deve preceder a atribuição de valores das variáveis `temp` e `temp_no`:*

*Se `ListaSimples.primeiro <> nulo` e
`ListaSimples.primeiro.prox <> nulo` então...*

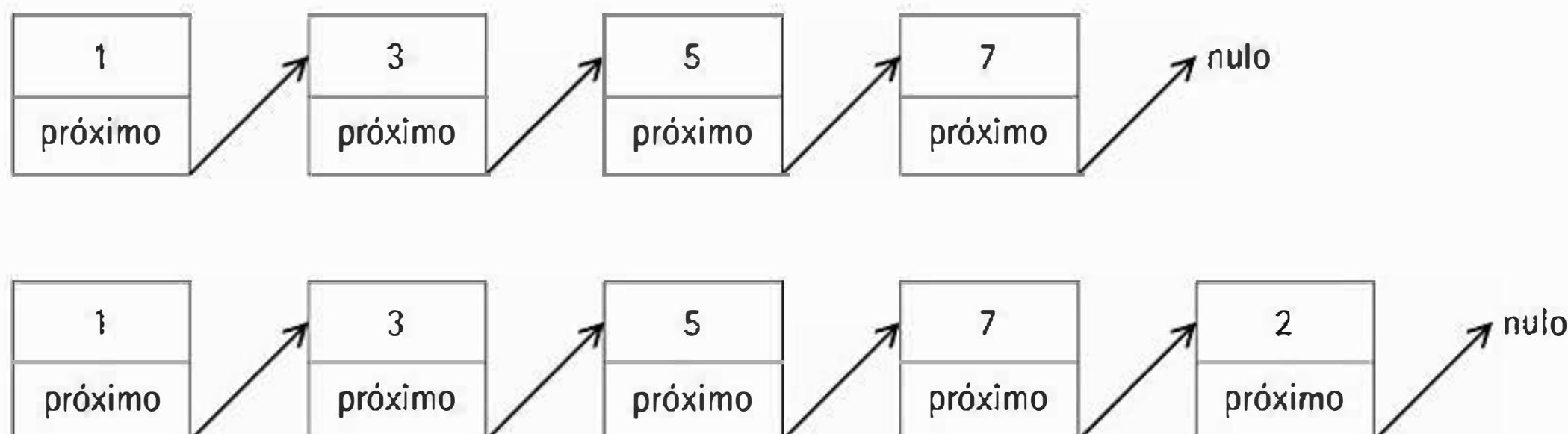
Na linha 32, é feito um teste de repetição, que utiliza a estrutura **Enquanto** para repetir as instruções enquanto o `prox`, do `temp_no`, for diferente de nulo ou `temp_no.valor` for diferente do elemento. Se o `temp_no.prox` for igual a nulo significa que o elemento lido é o último da lista e, se o valor do elemento for igual ao valor de `temp_no.valor`, significa que o elemento a ser excluído foi encontrado, então o laço de repetição é encerrado e são executadas as instruções `temp.prox ← temp_no^.prox`, por meio da qual o valor do próximo elemento do elemento a ser excluído é armazenado no próximo elemento do nó anterior ao que será excluído. **Lembre-se** de que o `temp` inicia com o valor do `primeiro` e o `temp_no`, com o valor do `primeiro.prox`. Com isso o elemento a ser excluído deixa de ser referenciado. Por último a instrução `desposicione(temp_no)` indica que o nó a ser excluído deixa definitivamente de ser referenciado — em algumas linguagens isso não precisa ser feito.

Vamos ver a seguir uma situação para utilização do algoritmo anterior. Suponha que a lista contenha os números inteiros 1, 3, 5, 7. Vamos inserir o número 2 na lista utilizando o procedimento `insereNo`.

O número 2 deverá ser passado como parâmetro para o procedimento e ficará armazenado na variável `novoNo`. Isso será feito no algoritmo principal da seguinte forma:

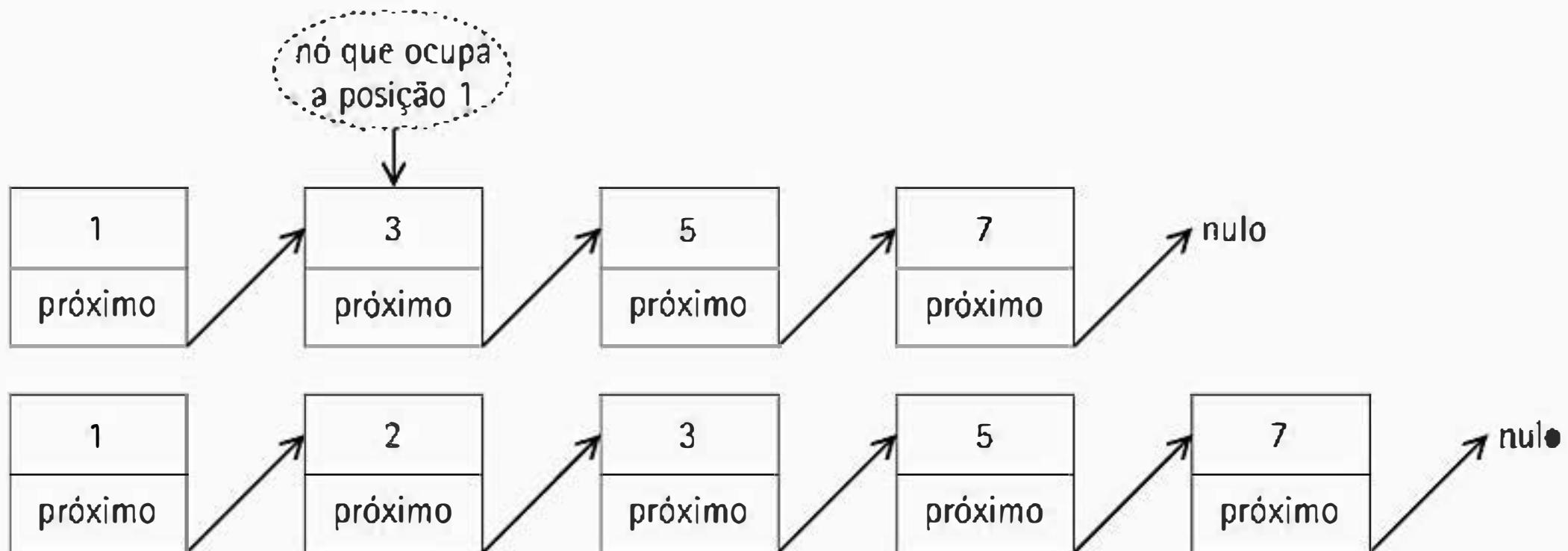
`InsereNo (2)`

então o procedimento é acionado e irá verificar se esse será o primeiro elemento da lista. Como no nosso exemplo isso é falso, pois a lista já contém os elementos 1, 3, 5 e 7, então será executada a instrução `ListaSimples.ultimo^.prox ← novoNo`, isto é, a referência do último elemento irá apontar para o `novoNo`, no caso o número 2, e a variável que representa o último elemento irá receber o `novoNo`.



|FIGURA 10.7| Inserção de um nó no final da lista (simples)

Nós também podemos inserir um novo nó em uma determinada posição da lista. Para isso, é preciso que sejam passados como parâmetros o valor a ser inserido e a posição que ele deverá ocupar, por exemplo: `incluirNo (2, 1)`.



[FIGURA 10.8] Inserção de um nó em uma posição específica da lista (simples)

A seguir serão apresentados os algoritmos para incluir nós em posições pré-determinadas:

A função `ContarNos` será utilizada para verificar a quantidade de nós existentes na lista. Ela não recebe parâmetros e retorna a quantidade de nós.

EXEMPLO 10.2: Pseudocódigo para representar uma função para contagem de nós de uma lista.

```

1. Função ContarNos( )
2.     var numero_nos: inteiro
3.     temp_no : NoSimples
4. Início
5.     numero_nos ← 0
6.     temp_no ← ListaSimples.primeiro
7.     Enquanto(temp_no^.prox <> nulo) faça
8.         numero_nos ← numero_nos + 1
9.         temp_no ← temp_no^.prox
10.    fim_enquanto
11.    return(numero_nos)

```

O procedimento `insereNo_posicao` recebe como parâmetros as variáveis `novoNo`, que é do tipo `NoSimples`, e `posicao`, que é do tipo `inteiro`.

EXEMPLO 10.3: Pseudocódigo para representar a inserção de um nó em uma posição específica em uma lista de encadeamento simples.

```

1. Procedimento InsereNo_posicao(var novoNo: NoSimples, posicao: inteiro)
2. var
3.     temp_no: NoSimples
4.     pos_aux: inteiro
5.     numero_nos: inteiro
6. Início

```

```

7.    pos_aux ← 0
8.    numero_nos ← ContarNos()
9.    Se (posicao = 0) então
10.       novoNo^.prox ← ListaSimples.primeiro
11.       ListaSimples.primeiro ← novoNo
12.       Se (ListaSimples.ultimo = ListaSimples.primeiro) então
13.          ListaSimples.ultimo ← novoNo
14.       fim-se
15.    Senão
16.       Se (posicao <= numero_nos) então
17.          Enquanto (temp_no^.prox <> nulo .ou. posicao <> pos_aux)
18.             temp_no ← temp_no^.prox
19.             pos_aux ← pos_aux + 1
20.         Fim-Enquanto
21.         novoNo^.prox ← temp_no^.prox
22.         temp_no^.prox ← novoNo
23.       Senão
24.          Se (posicao > numero_nos()) então
25.             ListaSimple.ultimo.prox ← novoNo
26.             ultimo ← novoNo
27.          fim-se
28.      fim-se
29.  fim-se
30. Fim

```

É verificado se a posição para inserção é 0 (linha 9), isto é, se o nó a ser inserido deverá ser o primeiro. Também é verificado se ele será o último elemento da lista (linha 12); se isso for verdadeiro, o valor do seu `prox` será nulo, o que significa que a lista só tem um elemento.

Se o novo nó tiver de ser inserido em outra posição, então é verificado se a posição é menor do que a quantidade de nós existentes na lista, o que é feito com o auxílio da função `ContarNos` construída anteriormente. Se isso for verdadeiro, então será utilizada uma estrutura de repetição (`Enquanto`) para encontrar o nó atual que ocupa a posição, depois é feito um deslocamento dos ponteiros, e para isso é utilizada uma variável que armazena temporariamente os valores do nó. Esse descolamento é feito nas linhas 21 e 22.

E por último é verificado se o nó deverá ser o último da lista, isto é, se a posição desejada é igual ao número de nós nas linhas 24 e 25. (Obs.: foi convencionado que qualquer valor para posição superior à quantidade de nós será o último da lista.)

Para implementarmos os algoritmos para criação e manipulação de listas de encadeamento simples em Java, precisaremos pensar um pouco diferente: nos algoritmos declaramos uma variável do tipo registro, que irá conter os elementos da lista (nós), declaramos uma variável que irá referenciar os elementos (apontador) e então declaramos a lista.

Em Java isso pode ser muito mais simples: basta criarmos uma classe que irá representar o nó, a qual deve conter a área de dados — no caso do exemplo, a variável do tipo inteiro e a variável que irá referenciar o próximo elemento, o apontador, no caso do exemplo `prox`.

EXEMPLO 10.4: Classe que cria a estrutura de um nó simples.

```

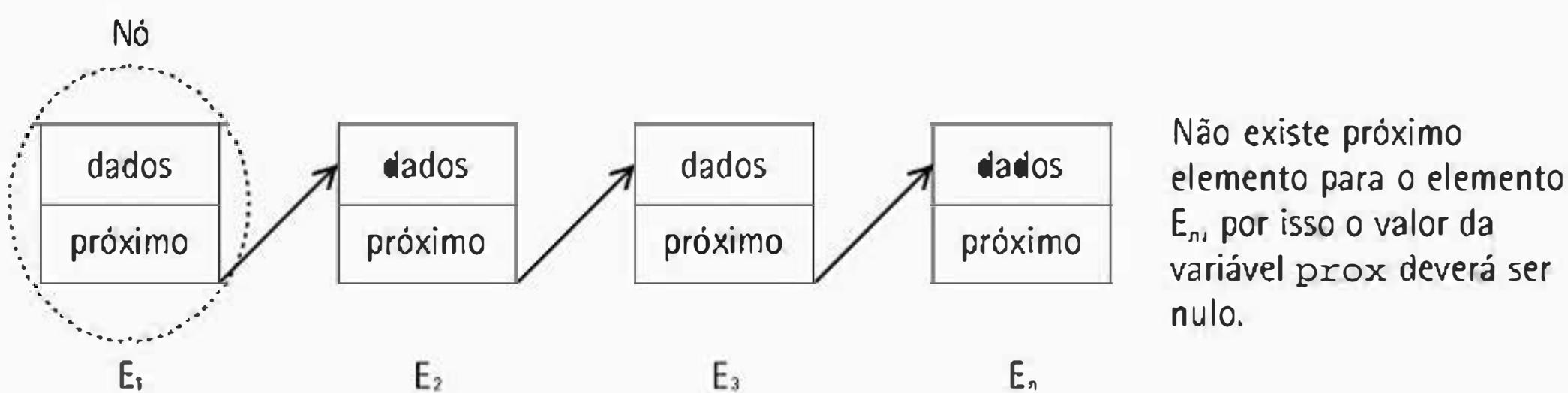
1. class IntNoSimples {
2.     int valor;
3.     IntNoSimples prox;
4.
5.     IntNoSimples(int ValorNo) {
6.         valor = ValorNo;
7.         prox = null;
8.     }
9. }
```

A classe `IntNoSimples` possui a declaração de duas variáveis: `valor`, que é do tipo inteiro, e `prox`, que é do tipo `IntNoSimples`. Observe que essa variável recebe o mesmo nome da classe, isso se faz necessário porque a variável `prox` é o campo que faz referência ao próximo elemento da lista. Como em Java não são aceitos ponteiros, essa variável deverá referenciar o próprio elemento, por isso deve possuir a mesma estrutura do nó.

LEMBRE-SE: Os programas escritos em Java podem ser reutilizados, então a classe `IntNoSimples` poderá ser utilizada por vários programas que precisem de um nó com encadeamento simples.

Essa classe também possui um método construtor chamado `IntNoSimples` (linha 5) que recebe como parâmetro passado por valor um valor para a variável `ValorNo`. Como já vimos no Capítulo 7, a passagem de parâmetro por valor não altera o valor original da variável, garantindo assim que o nó terá o seu valor preservado.

Na linha 7 a variável `prox` recebe um valor nulo, isso acontece porque quando um novo elemento é inserido na lista ele pode vir a ser o último elemento e, sendo assim, não possui próximo elemento. Veja na ilustração abaixo:



|FIGURA 10.9| Inserção de um nó no início da lista (simples)

EXEMPLO 10.5: A classe `ListaSimples101` cria a estrutura de uma lista, cujos nós terão a estrutura definida pela classe `IntNoSimples`. Neste exemplo os nós são inseridos sempre no final da lista.

```

1. class ListaSimples101{
2.     IntNoSimples primeiro, ultimo;
3.     int numero_nos;
4.
5.     ListaSimples101 (){
6.         primeiro = ultimo = null;
7.         numero_nos = 0;
8.     }
9.
10.    void insereNo_fim (IntNoSimples novoNo){
11.        novoNo.prox = null;
12.        if (primeiro == null)
13.            primeiro = novoNo;
14.
15.        if (ultimo != null)
16.            ultimo.prox = novoNo;
17.        ultimo = novoNo;
18.        numero_nos++;
19.    }
20. }
```

Na linha 2 as variáveis `primeiro` e `ultimo`, que representarão respectivamente o primeiro e o último elemento da lista, são definidas como do tipo `IntNoSimples`.

O construtor `ListaSimples101()`, linha 5, constrói uma lista vazia. Na linha 10 o método `insereNo_fim` recebe como parâmetro a variável `novoNo`, que é do tipo `IntNoSimples`; se esse for o primeiro elemento da lista, então a variável `primeiro` receberá o valor do `novoNo`; se já houver outros elementos ele será acrescentado no final da lista.

LEMBRE-SE: *O método construtor sempre leva o nome da classe.*

O programa `Exemplo101` acrescenta valores à lista. Na linha 3 é criado o objeto `Slist`, que é do tipo `ListaSimples101`. Nas linhas seguintes o método `insereNo_fim` é utilizado para inserir os valores na lista. Observe que o valor do nó é passado como parâmetro para o método `IntNoSimples` e ficará armazenado na variável `ValorNo`.

EXEMPLO 10.6: Programa em Java para utilizar a `ListaSimples101`.

```

1. class Exemplo101{
2.     public static void main(String[] args){
3.         ListaSimples101 Slist = new ListaSimples101 ();
4.         Slist.insereNo-fim (new IntNoSimples (1));
5.         Slist.insereNo-fim (new IntNoSimples (3));
6.         Slist.insereNo-fim (new IntNoSimples (5));
7.         Slist.insereNo-fim (new IntNoSimples (7));
8.     }
9. }
```

OBSERVAÇÃO: No Exemplo101, os valores são inseridos na lista sem o auxílio do usuário, já no Exemplo102 é o usuário quem digita os valores a serem inseridos na lista.

EXEMPLO 10.7: Programa completo para implementação de uma lista de encadeamento simples na qual são acrescentados os métodos para manipulação da lista: buscaNo, insereNo_inicio, insereNo_fim, insereNo_posicao, ContarNos e exibeLista. Confira no programa!

```

1. class ListaSimples102{
2.     IntNoSimples primeiro, ultimo;
3.     int numero_nos;
4.     ListaSimples102 (){
5.         primeiro = ultimo = null;
6.     }
7.     void insereNo_fim (IntNoSimples novoNo) {
8.         novoNo.prox = null;
9.         if (primeiro == null)
10.             primeiro = novoNo;
11.         if (ultimo != null)
12.             ultimo.prox = novoNo;
13.         ultimo = novoNo;
14.     }
15.     void insereNo_inicio (IntNoSimples novoNo) {
16.         if (primeiro != null)
17.             novoNo.prox = primeiro;
18.         else
19.             { if (primeiro == null)
20.                 primeiro = novoNo;
21.                 ultimo = novoNo;
22.             }
23.     }
24.     int ContarNos()
25.     { int tamanho = 0;
26.       IntNoSimples temp_no = primeiro;
27.       while (temp_no != null)
28.           { tamanho++;
29.             temp_no = temp_no.prox;
```

```
30.     }
31.     return tamanho;
32. }
33. void insereNo_posicao(IntNoSimples novoNo, int posicao)
34. {
35.     IntNoSimples temp_no = primeiro;
36.     int numero_nos = ContarNos();
37.     int pos_aux;
38.     if(posicao == 0)
39.         {novoNo.prox = primeiro;
40.          if(primeiro == ultimo)
41.              {ultimo = novoNo;}
42.          primeiro = novoNo;}
43.     else
44.         {if (posicao <= numero_nos)
45.          { pos_aux = 1;
46.            while(temp_no != null && posicao > pos_aux)
47.                {temp_no = temp_no.prox;
48.                 pos_aux ++;
49.                }
50.            novoNo.prox = temp_no.prox;
51.            temp_no.prox = novoNo;
52.          }
53.         else
54.             {if(posicao > numero_nos)
55.              {ultimo.prox = novoNo;
56.               ultimo = novoNo;
57.             } } } }
58. IntNoSimples buscaNo (int buscaValor){
59.     int i = 0;
60.     IntNoSimples temp_no = primeiro;
61.     while (temp_no != null)
62.         { if (temp_no.valor == buscaValor)
63.             {System.out.println("No " + temp_no.valor + "
64.                                         posição " + i);
65.              return temp_no;
66.            }
67.            i++;
68.            temp_no = temp_no.prox;
69.          }
70.     return null;
71. }
72. void excluiNo (int valor)
73. {
74.     IntNoSimples temp_no = primeiro;
75.     while (temp_no != null && temp_no.valor != valor)
76.         {temp_no = temp_no.prox;
77.          }
78.     temp_no.prox = temp_no.prox.prox;
79.     if (ultimo == temp_no.prox)
```

```

77.         ultimo = temp_no;
78.     }
79.     void exibeLista()
80.     {IntNoSimples temp_no = primeiro;
81.     int i = 0;
82.     while (temp_no != null)
83.         {System.out.println("Valor " + temp_no.valor + "
84.         posição " + i);
85.         temp_no = temp_no.prox;
86.         i++;}
86.     } } }
```

Alguns dos métodos apresentados no exemplo `ListaSimples102` não foram escritos no algoritmo `ExemploListaSimples`, por isso vamos discutir cada um dos métodos do exemplo `ListaSimples102`:

- `insereNo_fim` — esse método deve receber como parâmetro o valor que será incluído no final da lista; nesse caso esse valor está sendo representado pela variável `novoNo` do tipo `IntNoSimples` (linha 7). Observe que o valor de `prox` do `novoNo` deverá ser nulo e se a lista estiver vazia o primeiro elemento irá receber o `novoNo`. Se o último elemento for diferente de nulo, o `prox` (apontador) do último nó deverá fazer referência ao `novoNo` e então o valor do `novoNo` deverá ser armazenado na variável `ultimo`.

O método `insereNo_fim` não retorna nenhum valor, por isso é do tipo `void`.

- `insereNo_inicio` — esse método deve receber como parâmetro o valor que será incluído no início da lista; nesse caso esse valor está sendo representado pela variável `novoNo` do tipo `IntNoSimples` (linha 15). Deve-se verificar se já existe algum nó na lista, isto é, se a variável que representa o primeiro nó contém valores; se isso for verdadeiro, o `prox` do `novoNo` deverá fazer referência ao primeiro nó (linha 17) e armazenar na variável `primeiro` o valor do `novoNo`; caso contrário deverá ser verificado se o primeiro é nulo, isto é, se a lista estiver vazia, então as variáveis `primeiro` e `ultimo` receberão o valor do `novoNo`.

O método `inserNo_inicio` não retorna nenhum valor, por isso é do tipo `void`.

L.EMBRE-SE: Quando a lista está vazia ou contém apenas um nó o valor das variáveis que representam o primeiro e o último nós são iguais!

- `ContarNos` — esse método não recebe nenhum parâmetro, mas deve retornar um valor do tipo inteiro (linha 24). É utilizado para verificar a quantidade de nós que a lista contém.

- `insereNo_posicao` — esse método recebe dois parâmetros, um para `novoNo` do tipo `IntNoSimples` e outro para `posicao`, que é do tipo inteiro. A variável `posicao` representa a posição em que o nó deverá ser inserido no conjunto. Para determiná-la devemos verificar a quantidade de nós da lista com o método `ContarNos()` e então, se ela não for a primeira posição da lista, varrer a lista à procura da posição desejada (linha 45). Para isso é utilizado um nó auxiliar chamado `temp_no`. Cada vez que um nó é lido, deve-se indicar a leitura do próximo, então `temp_no` deverá receber o valor de `temp_no.prox` (linha 46), que traz a referência ao próximo nó e consequentemente o seu valor.

No caso de a posição desejada ser maior do que o número de nós, o `novoNo` será inserido como último nó da lista (linha 53).

Em todos os casos deverá ser feito o posicionamento do `novoNo` no conjunto e para isso, devem-se alterar as referências. Veja como exemplo `novoNo.prox = temp_no.prox; temp_no.prox = novoNo` (linhas 49 e 50).

Vamos seguir o mesmo exemplo apresentado no algoritmo do Exemplo 10.6: `insereNo_posicao(2, 1)` — o valor 2 deverá ser inserido na posição 1 da lista. O valor 2 estará representado pelo `novoNo` e o valor 1, pela variável `posicao`, então temos o esquema mostrado na Figura 10.10.

Quando o nó que ocupa a posição desejada (1) for encontrado, o laço de repetição será encerrado (linhas 45 a 48), então o valor que estará armazenado em `temp_no` será o valor do nó que ocupa a posição anterior à desejada e o `temp_no.prox` conterá a referência ao nó que está na posição desejada. Após o encerramento do teste de repetição é feita a alteração das referências dos nós para que o `novoNo` possa ser inserido:

- `novoNo.prox = temp_no.prox;` em nosso exemplo temos:
`novoNo.prox = a referência para o nó com valor 3;`

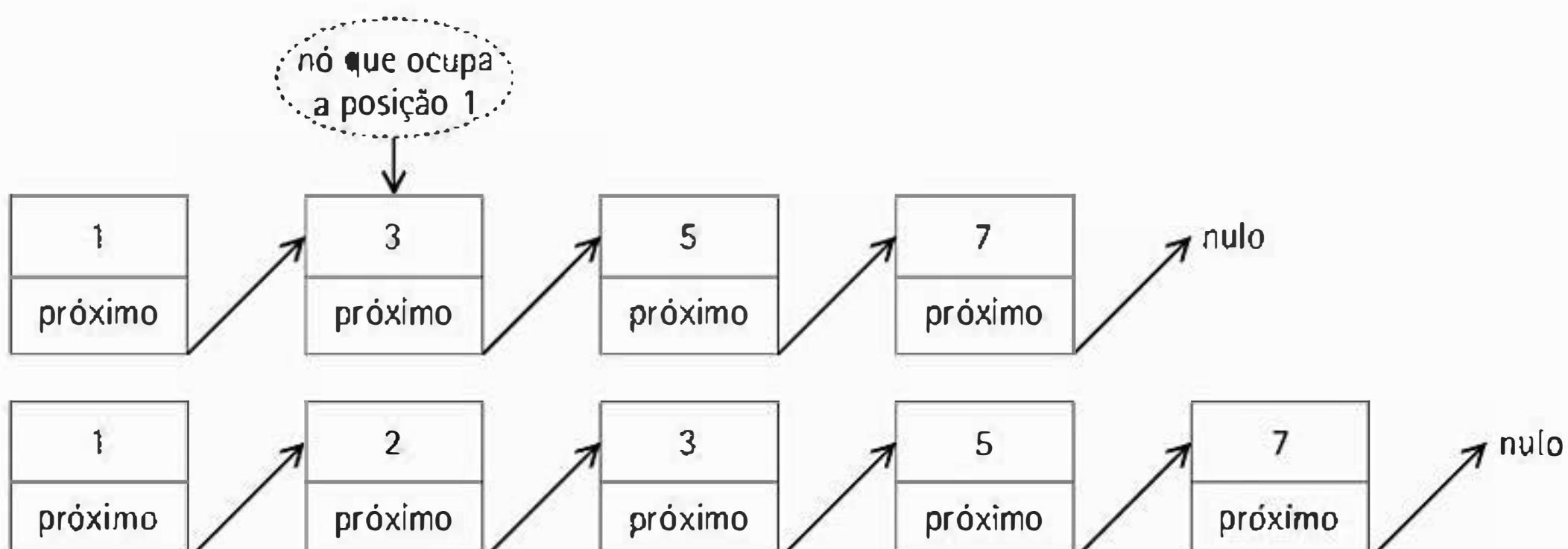


FIGURA 10.10 | Inserção de um nó em uma posição específica na lista (simples)

- `temp_no.prox = novoNo`, com isso o nó de valor 1 (que ocupa a posição 0) fará referência para o `novoNo` e este fará referência para o nó com valor 3.
- `buscaNo` — esse método recebe como parâmetro um valor inteiro para a variável `buscaValor`. Nós utilizamos a variável `temp_no` (novamente!) para nos auxiliar na localização do nó, para isso é feita a varredura de todos os nós da lista por meio de uma estrutura de repetição (linha 57) que compara `temp_no.valor` com a `buscaValor` e, quando encontra o nó, retorna o `temp_no`, caso contrário retorna nulo.
- `excluiNo` — esse método recebe como parâmetro um valor inteiro para a variável `valor`, que será o valor do nó a ser excluído. Nesse caso também utilizamos o `temp_no` e uma estrutura de repetição para percorrer a lista enquanto o valor não for igual ao `temp_no.valor` ou enquanto o `temp_no.prox` não for nulo. Quando o nó é encontrado, então é necessário que se deixe de fazer referência a ele, o que é feito na linha 75, onde o `temp_no.prox` recebe o valor do próximo elemento do próximo.
- O método `excluiNo` não retorna valores.
- `exibeLista` — esse método não recebe parâmetros e também não retorna valores. Ele exibe os nós da lista e, para isso, é utilizada a estrutura de repetição, que irá repetir o bloco para exibir o nó, enquanto não for encontrado nenhum nó nulo.

EXEMPLO 10.8: Com programa Exemplo102 poderemos utilizar todos os métodos da classe `ListaSimples102`.

```

1. import java.io.*;
2. class Exemplo102 {
3.     /* Constrói um menu de opções para manipulação da lista*/
4.     public static void escolhas () {
5.         System.out.print ("Escolha a Opção:");
6.         System.out.print ("\n1. Inserir Nó no início");
7.         System.out.print ("\n2. Inserir Nó no fim");
8.         System.out.print ("\n3. Inserir Nó em uma posição");
9.         System.out.print ("\n4. Localizar Nó");
10.        System.out.print ("\n5. Excluir Nó");
11.        System.out.print ("\n6. Exibir lista");
12.        System.out.print ("\n7. Sair");
13.        System.out.print ("Opcao :\t ");
14.    }
15.    public static void main(String args[]){
16.        ListaSimples102 Slist = new ListaSimples102();
17.        BufferedReader entrada;

```

```

17.     entrada = new BufferedReader(new
18.         InputStreamReader(System.in));
19.     int i = 0;
20.     IntNoSimples temp_no;
21.     int valor;
22.     try {
23.         escolhas();
24.         char opcao = entrada.readLine().charAt(0);
25.         while (opcao != '7') {
26.             switch (opcao) {
27.                 case '1' :
28.                     System.out.println("Inserir um Nó no inicio da
29. lista");
30.                     System.out.println("Digite um valor");
31.                     valor = Integer.parseInt(entrada.readLine());
32.                     Slist.insereNo_inicio(new IntNoSimples(valor));
33.                     break;
34.                 case '2' :
35.                     System.out.println("Inserir um Valor no final da
36. lista");
37.                     System.out.println("Digite um valor");
38.                     valor = Integer.parseInt(entrada.readLine());
39.                     Slist.insereNo_fim(new IntNoSimples(valor));
40.                     break;
41.                 case '3' :
42.                     System.out.println("Inserir um Valor numa dada
43. posicao");
44.                     System.out.println("Digite um valor");
45.                     valor = Integer.parseInt(entrada.readLine());
46.                     System.out.println("Digite a posicao");
47.                     int posicao = Integer.parseInt(entrada.readLine());
48.                     Slist.insereNo_posicao(new
49.                         IntNoSimples(valor),posicao);
50.                     break;
51.                 case '4':
52.                     System.out.println("Localiza um valor");
53.                     System.out.println("Digite um valor");
54.                     valor = Integer.parseInt(entrada.readLine());
55.                     Slist.buscaNo(valor);
56.                     break;
57.                 case '5':
58.                     System.out.println("Exclui um elemento da lista");
59.                     System.out.println("Digite o valor");
60.                     valor = Integer.parseInt(entrada.readLine());
61.                     Slist.excluiNo(valor);
62.                     break;
63.                 case '6':
64.                     System.out.println("Exibe a lista");
65.             }
66.         }
67.     }
68. 
```

```

60.         Slist.exibeLista();
61.         break;
62.     default : System.out.println ("Opcao Invalida!");
63. }
64. System.out.println();
65. escolhas();
66. opcao = entrada.readLine().charAt(0);
67. }
68. } catch (Exception erro){
69.     System.out.println ("Erro de Entrada de Dados");
70. }}}

```

10.3 LISTAS DUPLAMENTE ENCADEADAS

Quando se percorre uma lista de encadeamento simples é bastante difícil fazer o caminho inverso, já nas listas de encadeamento duplo esse problema não existe, pois cada nó possui uma referência para o próximo elemento da lista e outra para o anterior.

A concepção de uma lista duplamente encadeada é bastante similar à dos simples, basta acrescentar ao nó uma variável que fará referência ao elemento anterior, da mesma maneira que é feito com o próximo. Veja na linha 6 do ExemploListaDupla:

EXEMPLO 10.9: Pseudocódigo para representar uma lista duplamente encadeada.

```

1. Algoritmo ExemploListaDupla
2. Tipo apontador: ^NoDuplo
3.   NoDuplo = registro
4.           valor: inteiro
5.           prox: apontador
6.           ant: apontador
7.       fim
8.   ListaDupla = registro
9.           primeiro: apontador
10.          ultimo: apontador
11.          numero_nos: inteiro
12.      fim
13. Inicio
14.     ListaDupla.numero_nos ← 0
15.     ListaDupla.primeiro ← nulo
16.     ListaDupla.ultimo ← nulo

```

Para escrevermos algoritmos de manipulação de lista duplamente encadeada podemos seguir o mesmo raciocínio adotado para o entendimento da lista de encadeamento simples, mas devemos lembrar que o nó anterior também precisa ser referenciado a cada manipulação.

No algoritmo a seguir, que é continuação do ExemploListaDupla, demonstraremos os procedimentos para inserir um nó no final da lista e para excluir um nó de acordo com a sua posição.

Nesta solução para exclusão de nó é necessário descobrir qual é o nó que ocupa a posição desejada para a exclusão; para isso, utilizaremos a função pegarNo(). Essa função recebe como parâmetro o índice do nó a ser excluído e retorna o próprio nó.

```

17. Procedimento InsereNo_fim(var novoNo: NoDuplo)
18. inicio
19.     NovoNo^.prox ← nulo
20.     NovoNo^.ant ← ultimo
21.     if(ListaDupla.primeiro = nulo)
22.         ListaDupla.primeiro ← novoNo
23.     fim-se
24.     if{ListaDupla.ultimo < > nulo)
25.         ListaDupla.ultimo^.prox ← novoNo
26.     fim-se
27.     ListaDupla.ultimo ← novoNo
28.     ListaDupla.numero_nos ← ListaDupla.numero_nos + 1
29. fim
30. Procedimento pegarNo(var indice: inteiro): NoDuplo
31.     var
32.         temp_no: NoDuplo
33.         i: inteiro
34. inicio
35.     temp_no ← ListaDupla.primeiro
36.     Enquanto (temp_no < > nulo && i < = indice)
37.         temp_no ← temp_no^.prox
38.         i ← i + 1
39.     fim_enquanto
40.     Return temp_no
41. fim
42. Procedimento InsereNo_posicao(novoNo: NoDuplo, indice: inteiro)
43. var
44.     temp_no: NoDuplo
45. inicio
46.     temp_no ← pegarNo(indice)
47.     novoNo^.prox ← temp_no
48.     Se (temp_no^.prox < > nulo) então
49.         novoNo^.ant ← temp_no^.ant
50.         novoNo^.prox^.ant ← novoNo
51.     Senão
52.         novoNo^.ant ← novoNo
53.         ListaDupla .ultimo ← novoNo
54.     fim-se
55.     Se (indice = 0) então
56.         ListaDupla.primeiro ← novoNo

```

```

57.      Senão
58.          novoNo^ant^.prox ← novoNo
59.      fim-se
60.      ListaDupla.numero_nos ← ListaDupla.numero_nos + 1
61.  fim
62.  Procedimento excluiNo(indice: inteiro)
63.  var
64.      temp_no: NoDuplo
65.  inicio
66.      Se (indice = 0) então
67.          ListaDupla.primeiro ← ListaDupla.primeiro^.prox
68.          Se (ListaDupla.primeiro < > nulo) então
69.              ListaDupla.primeiro^.ant ← nulo
70.          fim-se
71.      Senão
72.          Se (temp_no < > ultimo) então
73.              temp_no ← pegarNo(indice)
74.              temp_no^.ant^.prox ← temp_no^.ant
75.      Senão
76.          ListaDupla.ultimo ← temp_no
77.      fim-se
78.  fim-se
79.  ListaDupla.numero_nos ← ListaDupla.numero_nos - 1
80.  fim
81. fim.

```

A seguir será apresentada a implementação da lista duplamente encadeada em Java. Observe que na classe `IntNoDuplo`, é criada a estrutura do nó, assim como fizemos para a lista simples, apenas acrescentamos o campo de referência `ant` que irá fazer referência ao nó anterior.

EXEMPLO 10.10: Classe que cria um nó duplo.

```

1. class IntNoDuplo{
2.     int valor;
3.     IntNoDuplo prox;
4.     IntNoDuplo ant;
5.
6.     IntNoDuplo (int ValorNo) {
7.         valor = ValorNo;
8.         prox = ant = null;
9.     }
10. }

```

EXEMPLO 10.11: Implementação dos mesmos métodos utilizados no algoritmo `ExemploListaDupla`.

```
1. class ListaDupla04{
2.     IntNoDuplo primeiro, ultimo;
3.     int numero_nos;
4.
5.     ListaDupla04 (){
6.         primeiro = ultimo = null;
7.         numero_nos = 0;
8.     }
9.
10.    void insereNo (IntNoDuplo novoNo){
11.        novoNo.prox = null;
12.        novoNo.ant = ultimo;
13.        if (primeiro == null)
14.            primeiro = novoNo;
15.        if (ultimo != null)
16.            ultimo.prox = novoNo;
17.        ultimo = novoNo;
18.        numero_nos++;
19.    }
20.    IntNoDuplo pegarNo (int indice){
21.        IntNoDuplo temp_no = primeiro;
22.        for (int i = 0; (i < indice) && (temp_no != null); i++)
23.            temp_no = temp_no.prox;
24.        return temp_no;
25.    }
26.    void incluiNo (IntNoDuplo novoNo, int indice){
27.        IntNoDuplo temp_no = pegarNo (indice);
28.        novoNo.prox = temp_no;
29.        if (novoNo.prox != null){
30.            novoNo.ant = temp_no.ant;
31.            novoNo.prox.ant = novoNo;
32.        } else {
33.            novoNo.ant = ultimo;
34.            ultimo = novoNo;
35.        }
36.        if (indice == 0)
37.            primeiro = novoNo;
38.        else
39.            novoNo.ant.prox = novoNo;
40.        numero_nos++;
41.    }
42.    void excluiNo (int indice){
43.        if (indice == 0){
44.            primeiro = primeiro.prox;
45.            if (primeiro != null)
46.                primeiro.ant = null;
47.        }else{
48.            IntNoDuplo temp_no = pegarNo (indice);
```

```
49.         temp_no.ant.prox = temp_no.prox;
50.         if (temp_no != ultimo)
51.             temp_no.prox.ant = temp_no.ant;
52.         else
53.             ultimo = temp_no;
54.     }
55.     numero_nos--;
56. }
57. }
```

EXEMPLO 10.12: Este programa utiliza as classes criadas no Exemplo 10.11.

```
1. class Exemplo104{
2.     public static void main(String[] args){
3.         ListaDupla104 Slist = new ListaDupla104 ();
4.         Slist.insereNo (new IntNoDuplo (1));
5.         Slist.insereNo (new IntNoDuplo (3));
6.         Slist.insereNo (new IntNoDuplo (5));
7.         Slist.insereNo (new IntNoDuplo (7));
8.         IntNoDuplo temp_no = Slist.primeiro;
9.         while (temp_no != null){
10.             System.out.println (temp_no.valor);
11.             temp_no = temp_no.prox;
12.         }
13.         Slist.incluiNo (new IntNoDuplo (2), 1);
14.         System.out.println ("Apos incluir o no 2...");
15.         temp_no = Slist.primeiro;
16.         while (temp_no != null){
17.             System.out.println (temp_no.valor);
18.             temp_no = temp_no.prox;
19.         }
20.         Slist.excluiNo (2);
21.         System.out.println ("Apos excluir o no 3...");
22.         temp_no = Slist.primeiro;
23.         while (temp_no != null){
24.             System.out.println (temp_no.valor);
25.             temp_no = temp_no.prox;
26.         }
27.     }
28. }
```

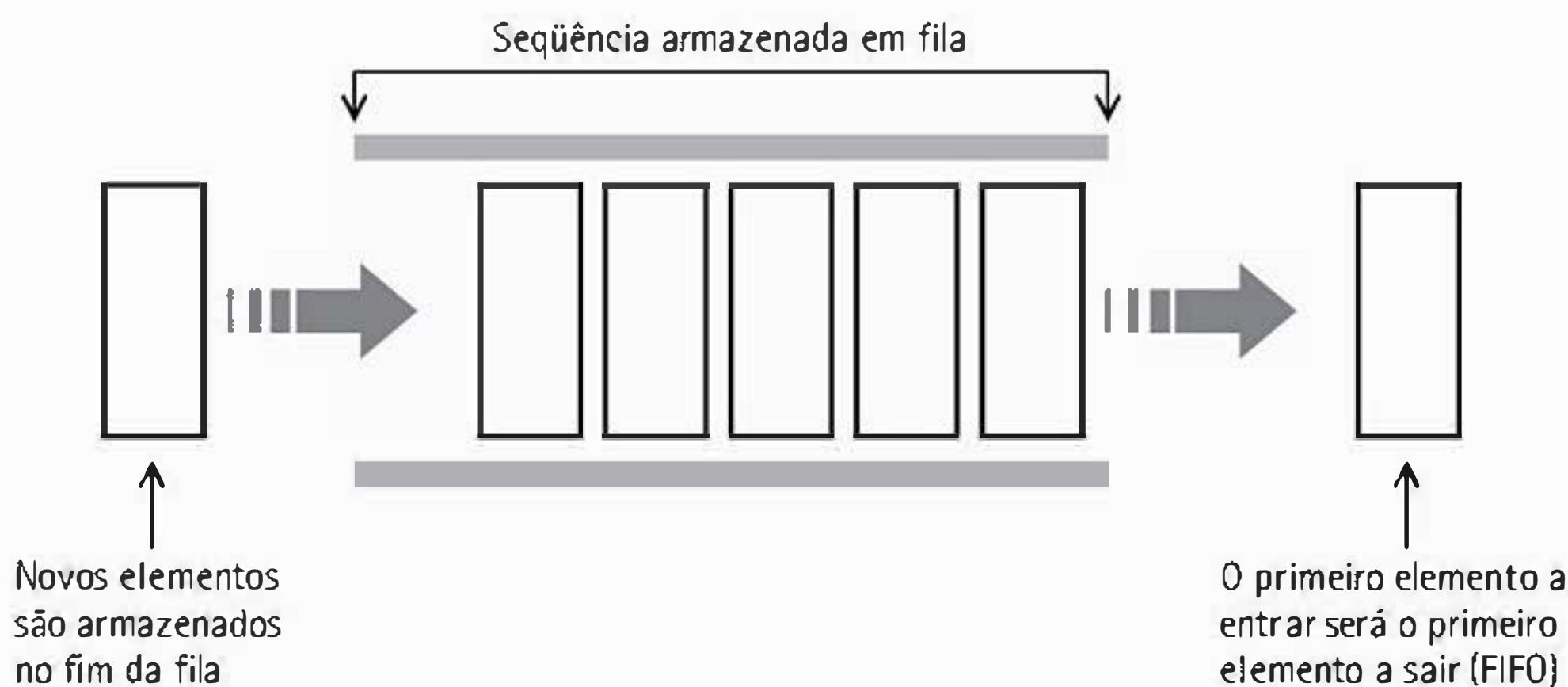
10.4 FILAS

Acho que todos nós já ficamos em uma fila. Fila para comprar ingressos para shows, pegar dinheiro no banco e, às vezes, até para comprar o pãozinho da manhã. O conceito de fila em programação é o mesmo dessas filas em que esperamos para ser atendidos em ordem: o primeiro elemento a entrar na fila será o primeiro elemento a

sair. Esse conceito é conhecido como ‘First In, First Out’ ou FIFO, expressão conhecida em português como PEPS ou ‘Primeiro Que Entra, Primeiro Que Sai’. Então, no conceito de fila, os elementos são atendidos, ou utilizados, seqüencialmente na ordem em que são armazenados.

As filas (*queues*) são conjuntos de elementos (ou listas) cujas operações de inserção são feitas por uma extremidade e de remoção, por outra extremidade.

Como exemplo pode-se implementar uma fila de impressão, em que os arquivos a ser impressos são organizados em uma lista e serão impressos na ordem de chegada, à medida que a impressora estiver disponível.



| FIGURA 10.11 | Conceito de fila

Conforme comentamos na introdução de listas, a implementação das listas, filas, pilhas e árvores pode ser feita por meio de arranjos ou de ponteiros. Até agora fizemos as implementações utilizando ponteiros, então iremos exemplificar a implementação de filas e pilhas por meio de arranjos, uma vez que o exemplo pode ser facilmente adaptado para o uso de ponteiros.

LEMBRE-SE:

Ao implementarmos a fila por meio de arranjos, estaremos utilizando um vetor como contêiner para o armazenamento dos elementos que estarão na fila.

Para definir a estrutura de uma fila, implementada por arranjo, é necessário construir um registro que contenha as informações da fila, como o início, o final e o contêiner de elementos, que é um vetor; nesse caso cada um dos elementos da fila será representado por uma posição no vetor. Veja a estrutura:

EXEMPLO 10.13: Pseudocódigo que representa uma fila implementada com arranjo.

1. Algoritmo Fila
2. var

```

3.    Tipo fila_reg = registro
4.          inicio: inteiro
5.          fim: inteiro
6.          elemento: vetor [1..50] de inteiro
7.      fim
8.    total: inteiro
9.    fila: fila_reg
10. inicio
11.     fila.inicio ← 0
12.     fila.fim ← 0
13.     total ← 0

```

Observe que a variável `elemento`, que é do tipo `vetor`, comporta 50 números inteiros. Essa característica é um limitador para trabalharmos com arranjos, pois podemos inserir apenas 50 valores!

NOTA:

Para implementar uma fila com o uso de ponteiros (alocação dinâmica), basta utilizar o nó e fazer as manipulações de acordo com o conceito PEPS.

Algoritmo que representa uma fila implementada com arranjo

```

14. Função vazia( ): lógica
15.     inicio
16.         Se(total = 0) entao
17.             return .v.
18.         Senão
19.             return .f.
20.         fim-se
21.     fim
22. Função cheia( ): lógica
23.     inicio
24.         Se(total >= 50) então
25.             return .v.
26.         Senão
27.             return .f.
28.         fim-se
29.     fim
30. Procedimento enfileirar(elem: inteiro)
31.     inicio
32.         Se (cheia( ) = .f..) então
33.             fila.elemento[inicio] ← elem
34.             fila.fim ← fila.fim + 1
35.             total ← total + 1
36.         Se (fila.fim >= 50) então
37.             fila.fim = 0
38.         fim-se

```

```

39.      Senão
40.          Mostre("Fila cheia!")
41.      fim-se
42.  fim
43. Funcao desenfileirar( ): inteiro
44.  var
45.      excluido: inteiro
46.  inicio
47.      Se (vazia( ) = .f.) então
48.          excluido ← fila.elemento[inicio]
49.          fila.inicio ← fila.inicio + 1
50.          Se (fila.inicio > = tamanho) então
51.              fila.inicio ← 0
52.          fim-se
53.          total ← total - 1           retorna excluido
54.      Senão
55.          excluido ← nulo
56.          retorna excluido
57.      fim-se
58.  fim
59. Procedimento exibeFila( )
60.  var
61.      i: inteiro
62.  inicio
63.      Para(i ← 0 até total) faça
64.          Mostre("Posição ", i, " valor ", elemento[i])
65.      fim-para
66.  fim
67. fim.

```

A seguir será apresentado o programa escrito em Java para representar esse algoritmo, então faremos as explicações do código, assim evitaremos a repetição das explicações, que são similares para o código e para o algoritmo!

Java: classe que representa uma fila implementada com arranjo

```

1. class Fila {
2.     int tamanho;
3.     int inicio;
4.     int fim;
5.     int total;
6.     Object vetor[];
7.     Fila(int tam) {
8.         inicio = 0;
9.         fim = 0;
10.        total = 0;
11.        tamanho = tam;

```

```
12.         vetor = new Object[tam];
13.     }
14.     public boolean vazia () {
15.         if (total == 0)
16.             return true;
17.         else
18.             return false;
19.     }
20.     public boolean cheia () {
21.         if (total >= tamanho)
22.             return true;
23.         else
24.             return false;
25.     }
26.     public void enfileirar(Object elem) {
27.         if (!cheia())
28.             { vetor[fim] = elem;
29.               fim++;
30.               total++;
31.               if (fim >= tamanho)
32.                   fim = 0;
33.             }
34.         else
35.             { System.out.println("Fila Cheia");
36.             }
37.     }
38.     public Object desenfileirar()
39.         { Object excluido;
40.             if (vazia() == false)
41.                 {excluido = vetor[inicio];
42.                  inicio++;
43.                  if (inicio >= tamanho)
44.                      inicio = 0;
45.                  total--;
46.                  return excluido;
47.                 } else
48.                     { excluido = null;
49.                       return excluido;
50.                     }
51.                 }
52.             }
53.             public void exibeFila()
54.                 {for (int i = 0; i < total; i++)
55.                  {System.out.println("posicao " + i + " valor " +
vetor[i]);
56.                  }
57.                 }
58. }
```

Para representarmos uma fila em Java criamos a classe `Fila`, que contém as variáveis para tamanho da fila, início e fim da fila, total de elementos da fila e o vetor para armazenamento dos elementos, linhas 1 a 6. Observe que o vetor irá armazenar elementos do tipo objeto; nesse caso, podem ser armazenados quaisquer tipos de valores e essa fila pode ser considerada uma fila genérica. Observe também que o tamanho do vetor é passado como parâmetro para o método construtor, linha 7.

Vamos discutir cada um dos métodos que foram apresentados no programa `Fila`:

- `vazia()` e `cheia()` — linhas 14 e 22, não recebem nenhum parâmetro e retornam um valor lógico. Esses métodos são utilizados para verificar se a fila está vazia ou cheia. A verificação é feita comparando-se a variável `total` com 0 ou com a quantidade de elementos que o vetor comporta, no caso do exemplo apresentado para `vazia` (linha 16) Se o total for igual a 0, significa que a fila está vazia e o método retornará `verdadeiro`, caso contrário retornará `falso`. Já para verificar se a fila está cheia, na linha 24 o teste feito compara o total com a quantidade de elementos que o vetor comporta (`tamanho`), então se isso for `verdadeiro` significa que a fila está cheia e o retorno é `verdadeiro`, caso contrário o retorno é `falso`.

Essas funções serão utilizadas nas chamadas aos métodos para enfileirar ou desenfileirar elementos. Elas são importantes para que seja verificado se essas operações serão ou não possíveis de ser realizadas.

- `enfileirar` — linha 26, recebe como parâmetro um valor do tipo `Object` para a variável `elem` e não retorna nenhum valor, por isso é do tipo `void`. Na linha 27 é feita uma chamada ao método `cheia` para verificar se é possível a inserção de um novo elemento na fila. A expressão `!cheia()` é similar a `cheia() == false`. O elemento deve ser inserido no final da fila, a fim de que seja respeitado o conceito PEPS, então o último elemento do vetor deverá ser o elemento que está sendo inserido: `vetor[fim] = elem` (linha 28). As variáveis `fim` e `total` devem ser incrementadas, então é verificado se a variável `fim` é maior ou igual a `tamanho`, para garantir que a fila contenha a quantidade de elementos que podem ser comportados pelo vetor — esse recurso força a circularidade da fila. Veja o esquema a seguir:

Suponha que a fila tenha capacidade para cinco elementos do tipo inteiro:

posição	0	1	2	3	4
valor	45	7	9	32	1
organização	primeiro elemento				último elemento

|FIGURA 10.12| Representação de uma fila

Se retirarmos um elemento, o elemento a ser retirado será o que está em primeiro lugar na fila, no caso ele ocupa a posição 0 e seu valor é 45. Com a retirada do elemento a fila ficará da seguinte maneira:

posição	1	2	3	4
valor	7	9	32	1
organização	primeiro elemento Início			último elemento Fim

|FIGURA 10.13| Representação da fila após a remoção de um elemento

O elemento que passou a ser o primeiro da fila ocupa a posição 1 no vetor e o seu valor é 7. Observe que ainda cabe um elemento. Conforme os elementos vão sendo retirados da fila, o início da fila deve ser incrementado, de maneira que o primeiro elemento do conjunto a entrar na fila seja sempre o primeiro a sair.

Quando o tamanho da fila, que determina o número de posições que o vetor pode conter, estiver completo, se existirem posições vazias os elementos inseridos deverão ocupar as posições ‘desocupadas’ que estão no início do vetor, por isso o fim da lista é zerado (linhas 31 e 32), sempre respeitando a ordem imposta por PEPS. Para isso o início e o fim devem ser incrementados.

posição	1	2	3	4	0
valor	7	9	32	1	novo valor
organização	primeiro elemento Início			último elemento	Fim

|FIGURA 10.14| Representação do fim da fila

Quando um novo valor for inserido será armazenado na posição 0, que é o final da fila, e o último elemento será este!

- **desenfileirar** — o método **desenfileirar** não recebe nenhum parâmetro, pois o elemento a ser excluído é o elemento que está no início da fila. Esse método retorna o elemento que está sendo excluído, que é do tipo **Object**, para isso foi declarada a variável **excluido**, na linha 39. Nesse caso, se não existir nenhum valor na fila, o retorno do método será **null**; isso é possível pois o dado é do tipo **Object**, que aceita nulos.

OBSERVAÇÃO: No algoritmo também atribuímos **null** ao retorno, somente a título de representação, pois o tipo de dado que a função retorna é **inteiro**, e este não aceita nulos.

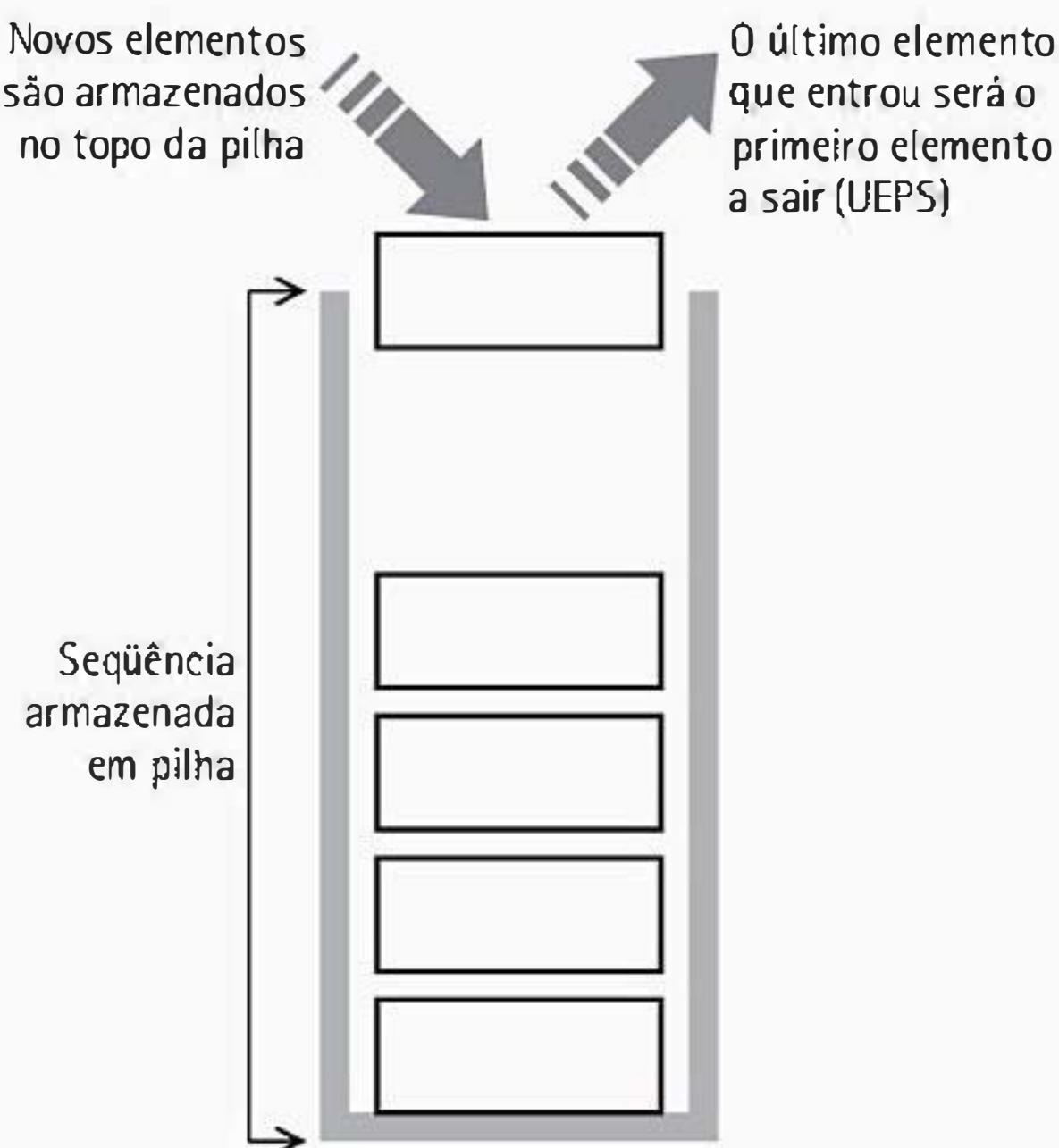
Na linha 40 é feita uma chamada ao método **vazia()**, cujo resultado será testado. A instrução **vazia() == false** é similar a **!vazia()**, então, se a fila não estiver vazia a variável **excluido** receberá o primeiro elemento da fila (linha 41) e o início da fila deverá ser incrementado (linha 41) e o total de elementos, decrementado (linha 45). Note que, na linha 43, se **inicio** for


```

34.                     {System.out.println
35.             (objFila.desenfileirar());
36.         }
37.         else
38.             {System.out.println("Fila Vazia!");
39.         }
40.         break;
41.     case '3' :
42.         objFila.exibeFila();
43.         break;
44.     default : System.out.println ("Opcao Invalida!");
45.     }
46.     System.out.println();
47.     escolhas();
48.     opcao = entrada.readLine().charAt(0);
49. }
50. catch (Exception erro){
51.     System.out.println ("Erro de Entrada de Dados");
52. }
53. }
```

10.5 PILHAS

As pilhas também são conhecidas como lista LIFO (Last In, First Out), que em português significa ‘último a entrar e primeiro a sair’ (UEPS). É uma lista linear em que



|FIGURA 10.15| Conceito de pilha

todas as operações de inserção e remoção são feitas por um único extremo denominado topo. Um exemplo bastante comum em que se aplica o conceito de pilhas é o de uma pilha de pratos que estão guardados no armário: quando a pessoa vai utilizar um deles pega sempre o prato que se encontra no topo da pilha, assim como, quando um novo prato vai ser guardado, é colocado no topo. Isso acontece porque apenas uma das extremidades da pilha está acessível.

LEMBRE-SE: A operação de inserção é denominada empilhamento e a de exclusão, desempilhamento.

EXEMPLO 10.14: Pseudocódigo que representa uma pilha implementada com arranjo.

```

1. Algoritmo Pilha
2.   var
3.     Tipo pilha_reg = registro
4.       topo: inteiro
5.       elemento: vetor[1..50] de inteiros
6.     fim
7.     pilha: pilha_reg
8. inicio
9.   pilha.topo ← -1
10.  Função vazia( ): lógica
11.    início
12.      Se (pilha.topo = -1) então
13.        retorne .v.
14.      Senão
15.        retorne .f.
16.      fim-se
17.    fim
18.  Função cheia( ): lógica
19.    início
20.      Se (pilha.topo >= 50) então
21.        retorne .v.
22.      Senão
23.        retorne .f.
24.      fim-se
25.    fim
26.  Procedimento empilhar{elem: inteiro}
27.    início
28.      Se (cheia( ) = .f.) então
29.        pilha.topo ← pilha.topo + 1
30.        elemento.topo ← elem
31.      Senão
32.        Mostre("Pilha Cheia!")
33.      fim-se
34.  Função desempilhar( ): inteiro
35.    var

```

```

36.         valorDesempilhado : inteiro
37.         início
38.             Se (vazia ( ) = .f.) então
39.                 Mostre("Pilha vazia!")
40.                 valorDesempilhado ← nulo
41.                 retorno(valorDesempilhado)
42.             Senão
43.                 valorDesempilhado ← pilha.vetor[topo]
44.                 pilha.topo ← pilha.topo - 1
45.                 retorno(valorDesempilhado)
46.             fim-se
47.         Procedimento exibePilha( )
48.             var
49.                 i : inteiro
50.             início
51.                 Para(i ← 0 até topo) faça
52.                     Mostre("Elemento ", elemento[i], " posição ", i)
53.                 Fim-para
54.             fim
55.         fim.

```

A definição da estrutura da pilha é bastante similar à definição de uma fila. Observe que na fila temos o **início**, o **fim** e o vetor; na pilha temos o **topo** e o vetor (linhas 3 a 7). Como na pilha as inserções e remoções são feitas por uma única extremidade, denominada **topo**, não é necessário que se conheça o elemento da outra extremidade; já na fila isso se faz necessário, uma vez que as inserções são feitas por uma extremidade (**fim**) e as remoções, por outra (**início**).

NOTA:

Para implementar uma pilha utilizando alocação dinâmica (ponteiros), basta utilizar a estrutura do nó e fazer as manipulações de acordo com o conceito UEPS.

A seguir vamos apresentar o programa em Java para implementar uma pilha e então faremos os comentários sobre o código.

Java: classe que representa uma pilha

```

1. class Pilha {
2.     int tamanho;
3.     int topo;
4.     Object vetor[];
5.     Pilha(int tam) {
6.         topo = -1;
7.         tamanho = tam;
8.         vetor = new Object[tam];
9.     }
10.    public boolean vazia ()

```

```

11.         {if (topo == -1)
12.             return true;
13.         else
14.             return false;
15.     }
16.     public boolean cheia ()
17.     {if (topo >= tamanho)
18.         return true;
19.     else
20.         return false;
21. }
22.     public void empilhar(Object elem)
23.     {if (cheia( ) == false)
24.         {topo++;
25.          vetor[topo] = elem;
26.        }
27.     else
28.         {System.out.println("Pilha Cheia");
29.        }
30.     }
31.     public Object desempilhar()
32.     {Object valorDesempilhado;
33.     if (vazia( ) == false)
34.         { System.out.print("Pilha Vazia");
35.         valorDesempilhado = null;
36.         return valorDesempilhado;
37.       }
38.     else
39.         { valorDesempilhado = vetor[topo];
40.           topo--;
41.           return valorDesempilhado;
42.         }
43.     }
44.     public void exibePilha()
45.     {for(int i = topo; i >= 0; i--)
46.      System.out.println("Elemento " + vetor[i] + " posicao " +
i);
47.    }
48. }

```

No algoritmo criamos um registro para representar a pilha, no programa em Java devemos criar uma classe. Na definição do vetor, no algoritmo, determinamos o tamanho máximo de elementos aceitos: elemento: vetor [1.. 50] de inteiros. Já na classe podemos definir o vetor: Object vetor [], e no método construtor da classe passar como parâmetro o tamanho do vetor: vetor = new Object(tam).

Vamos discutir cada um dos métodos que foram apresentados no programa Fila:

- **vazia()** e **cheia()** — linhas 10 e 16, não recebem nenhum parâmetro e retornam um valor lógico. Esses métodos são utilizados para verificar se a pilha está vazia ou cheia. A verificação é feita comparando-se a variável **topo** com -1 ou com a quantidade de elementos que o vetor comporta, no caso do exemplo apresentado para **vazia** (linha 10) Se o topo for igual a -1, significa que a pilha está vazia e retornará **verdadeiro**, caso contrário retornará **falso**. Já para verificarmos se a pilha está cheia, na linha 16, o teste feito compara o topo com a quantidade de elementos que o vetor comporta (**tamanho**), então se isso for verdadeiro significa que a pilha está cheia e o retorno é **verdadeiro**, caso contrário o retorno é **falso**.

Essas funções serão utilizadas nas chamadas aos métodos para empilhar ou desempilhar elementos. Elas são importantes para que seja verificado se essas operações serão ou não possíveis de ser realizadas.

- **empilhar** — linha 22, recebe como parâmetro um valor do tipo **object** para a variável **elem** e não retorna nenhum valor, por isso é do tipo **void**. Na linha 23 é feita uma chamada ao método **cheia** para verificar se é possível a inserção de um novo elemento na pilha. O elemento deve ser inserido no topo da pilha, a fim de que o conceito UEPS seja respeitado, então o elemento do topo deverá ser o elemento que está sendo inserido: **vetor[topo] = elem** (linha 25), e a variável **topo** deve ser incrementada.

Veja o esquema a seguir.

Suponha que a pilha tenha capacidade para cinco elementos do tipo inteiro, então vamos inserir o elemento 45:

posição	valor
0	45

topo

|FIGURA 10.16| Inserção de um elemento na pilha

Se inserirmos um novo elemento, por exemplo o número 7, a pilha ficará da seguinte maneira:

posição	valor
1	7
0	45

topo

|FIGURA 10.17| Representação da pilha após a inserção de um novo elemento

Se continuarmos as inserções até que a pilha fique cheia teremos:

posição	valor
4	1
3	32
2	9
1	7
0	45

topo

|FIGURA 10.18| Representação da pilha completamente preenchida

Observe que o último valor que foi inserido, o número 1, é o valor que ocupa o topo da pilha, já o primeiro valor que foi inserido, o número 45, fica no final da pilha e, nesse caso, será o último elemento a ser retirado da pilha (desempilhado).

posição	valor
4	
3	32
2	9
1	7
0	45

topo

|FIGURA 10.19| Representação da pilha após uma remoção

Observe que o elemento seguinte passou a representar o topo.

IMPORTANTE!

Em nossa representação deixamos de mencionar o número 1, que estava no topo, mas na verdade essa é uma operação lógica, e não física. A referência ao topo passa para o elemento seguinte, com isso o valor que estava no topo deixa de ser referenciado na pilha e o seu espaço poderá ser ocupado por outro valor.

- **desempilhar** — o método desempilhar não recebe nenhum parâmetro, pois o elemento a ser excluído é o elemento que está no topo da pilha. Esse método retorna o elemento que está sendo excluído, que é do tipo Object, e para isso foi declarada a variável valorDesempilhado, na linha 32. Nesse caso, se não existir nenhum valor na fila o retorno do método será null, isso é possível pois o dado é do tipo Object, que aceita nulos.

OBSERVAÇÃO:

No algoritmo também atribuímos nulo ao retorno, somente a título de representação, pois o tipo de dado que a função retoma é inteiro, e este não aceita nulos.

Na linha 33 é feita uma chamada ao método `vazia()`, cujo resultado será testado. Então, se a pilha não estiver vazia, a variável `valorDesempilhado` receberá o elemento do topo da pilha (linha 39) e o topo da pilha deverá ser decrementado, conforme visto no exemplo anterior.

- `exibePilha` — este método não recebe nenhum parâmetro e também não retorna nada. É utilizado para exibir os elementos da pilha, o que é feito com o uso da estrutura de repetição `for`, que repete o processo de exibição do valor do elemento e da posição que ele ocupa até que a variável de controle `i` atinja o valor do `total` de elementos.

A classe `usaPilha` possibilita que o usuário selecione a opção desejada para manipulação da pilha e utiliza todos os métodos que implementamos na classe `Pilha`.

Java: classe que utiliza uma pilha

```
1. import java.io.*;
2. class usaPilha {
3.     public static void escolhas () {
4.         System.out.print ("Escolha a Opcão:");
5.         System.out.print ("\n1. Inserir");
6.         System.out.print ("\n2. Excluir");
7.         System.out.print ("\n3. Exibir a Pilha");
8.         System.out.print ("\n4. Sair");
9.         System.out.print ("\n. Opcão :\t ");
10.    }
11.    public static void main(String args[]){
12.        Pilha objPilha = new Pilha(10);
13.        BufferedReader entrada;
14.        entrada = new BufferedReader(
15.            new InputStreamReader (System.in));
16.        Object valor;
17.        try {
18.            escolhas();
19.            char opcao = entrada.readLine().charAt(0);
20.            while (opcao != '4') {
21.                switch (opcao) {
22.                    case '1' :
23.                        if (objPilha.cheia() == false)
24.                            {System.out.print ("Digite algo: ");
25.                            valor = entrada.readLine();
26.                            objPilha.empilhar(valor);
27.                            }
28.                        else
29.                            {System.out.println("Fila Cheia!");
30.                            }
31.                        break;
32.                    case '2' :
```

```

33.             if (objPilha.vazia() == false)
34.                 {System.out.println (objPilha.desempilhar());
35. }
36.             else
37.                 {System.out.println("Fila Vazia!");
38. }
39.             break;
40.         case '3' :
41.             objPilha.exibePilha();
42.             break;
43.         default : System.out.println ("Opcao Invalida!");
44.     }
45.     System.out.println();
46.     escolhas();
47.     opcao = entrada.readLine().charAt(0);
48. }
49. } catch (Exception erro){
50.     System.out.println ("Erro de Entrada de Dados");
51. }
52. }
53. }
```

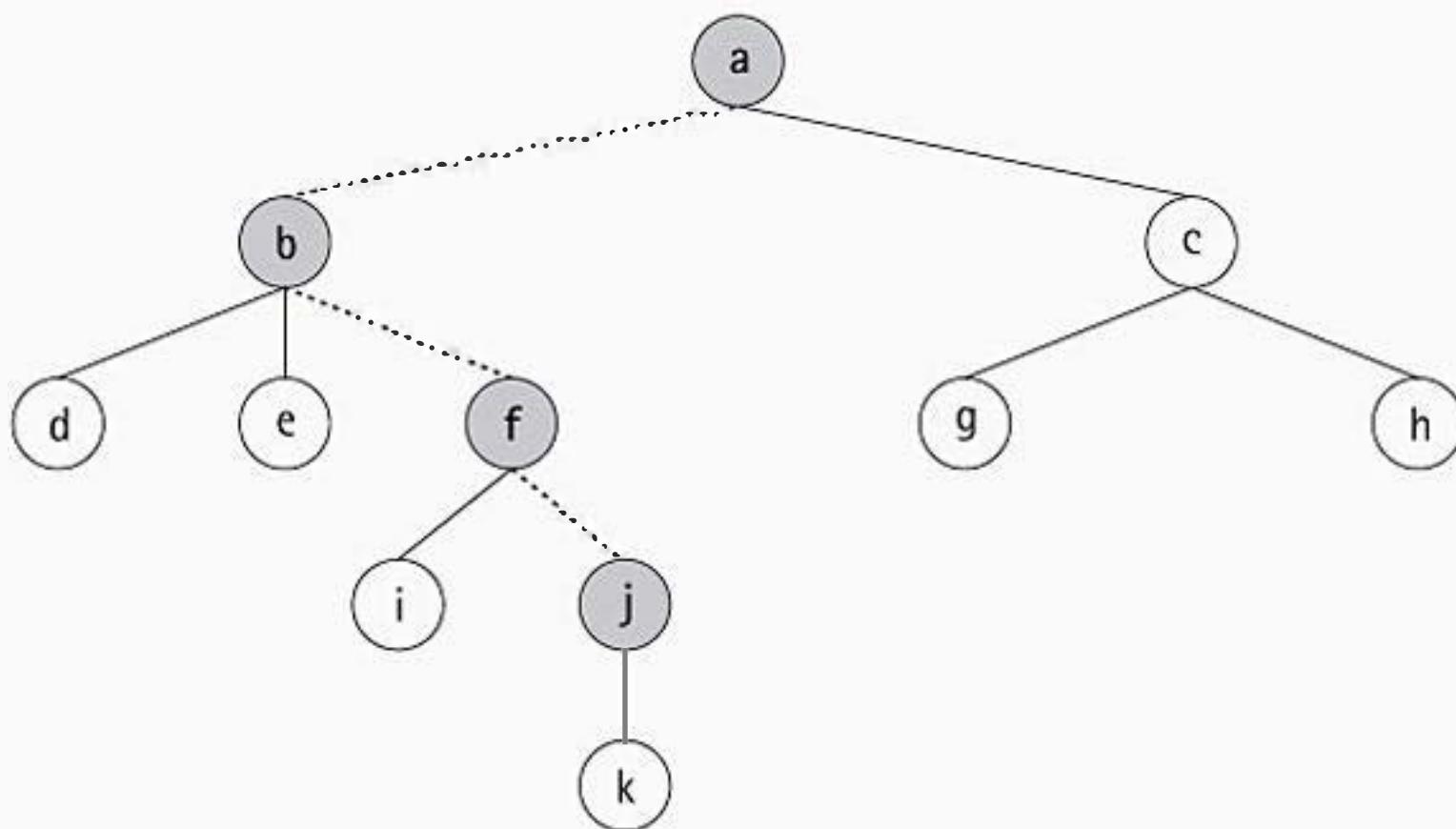
10.6 ÁRVORES

Uma árvore é uma estrutura de dados bidimensional, não linear, que possui propriedades especiais e admite muitas operações de conjuntos dinâmicos, tais como: consulta, inserção, remoção, entre outros. É diferente das listas e pilhas, uma vez que estas são estruturas de dados lineares.

Uma árvore, de modo geral, possui as seguintes características:

- **nó raiz** — nó do topo da árvore, do qual descendem os demais nós. É o primeiro nó da árvore;
- **nó interior** — nó do interior da árvore (que possui descendentes);
- **nó terminal** — nó que não possui descendentes;
- **trajetória** — número de nós que devem ser percorridos até o nó determinado;
- **grau do nó** — número de nós descendentes do nó, ou seja, o número de subárvore de um nó;
- **grau da árvore** — número máximo de subárvore de um nó;
- **altura da árvore** — número máximo de níveis dos seus nós;
- **altura do nó** — número máximo de níveis dos seus nós.

Para exemplificar a explicação sobre as características de uma árvore, vamos fazer uma análise da árvore apresentada na Figura 10.20:



[FIGURA 10.20] Árvores

- o nó a é denominado nó raiz, tem grau dois pois possui dois filhos, os nós b e c, que também podem ser chamados de subárvores ou nós descendentes;
- o nó b tem grau três pois possui três filhos: os nós d, e e f; o nó b também é denominado pai dos nós d, e e f;
- os nós d e e são nós terminais, isto é, não possuem descendentes e por isso têm grau zero;
- o nó f tem grau dois e tem como filhos os nós i e j;
- o nó i é um nó terminal e possui grau zero;
- o nó j tem grau um e é pai do nó k, que é terminal;
- o nó c tem grau dois e é pai dos nós g e h, que são nós terminais;
- a árvore possui grau três, pois este é o número máximo de nós descendentes de um único pai;
- a árvore tem altura igual a 5, já o nó b tem altura igual a 4, o nó c tem altura igual a 2, o nó k tem altura igual a 1 e assim por diante;
- para definirmos a trajetória a ser percorrida vamos supor que se deseje chegar ao nó j, então o caminho a ser percorrido seria a, b, f, j, conforme ilustrado na Figura 10.21.

As árvores podem ser do tipo listas generalizadas ou binárias. As árvores do tipo listas generalizadas possuem nós com grau maior ou igual a zero, enquanto uma árvore do tipo binária sempre possui nós com grau menor ou igual a 2. Veja os exemplos de árvores apresentados na Figura 10.22.

Neste livro trataremos apenas das árvores binárias.

10.6.1 ÁRVORES BINÁRIAS

Conforme já dissemos anteriormente, uma árvore binária sempre possui nós com grau menor ou igual a dois, isto é, nenhum nó possui mais do que dois descendentes

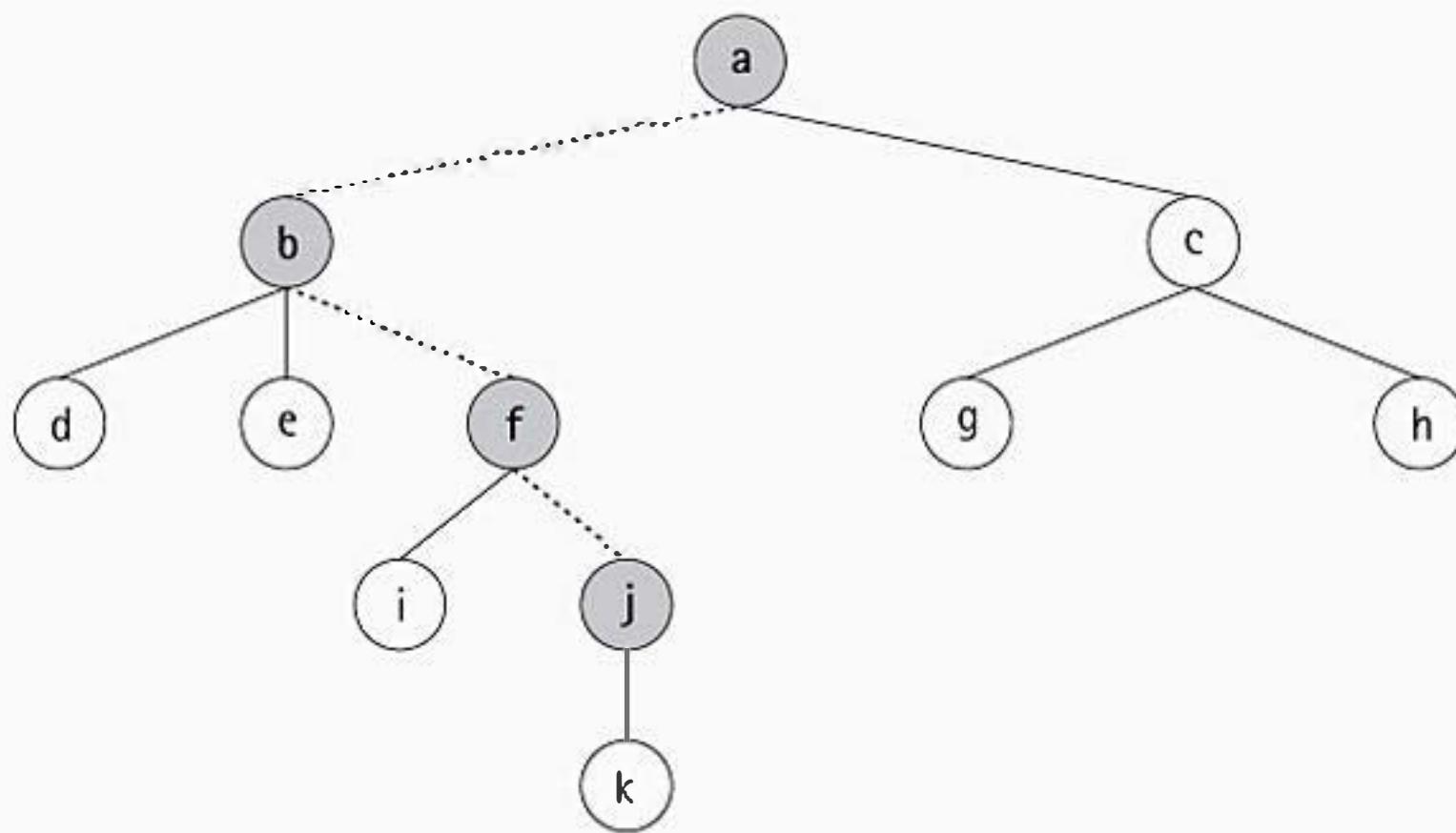
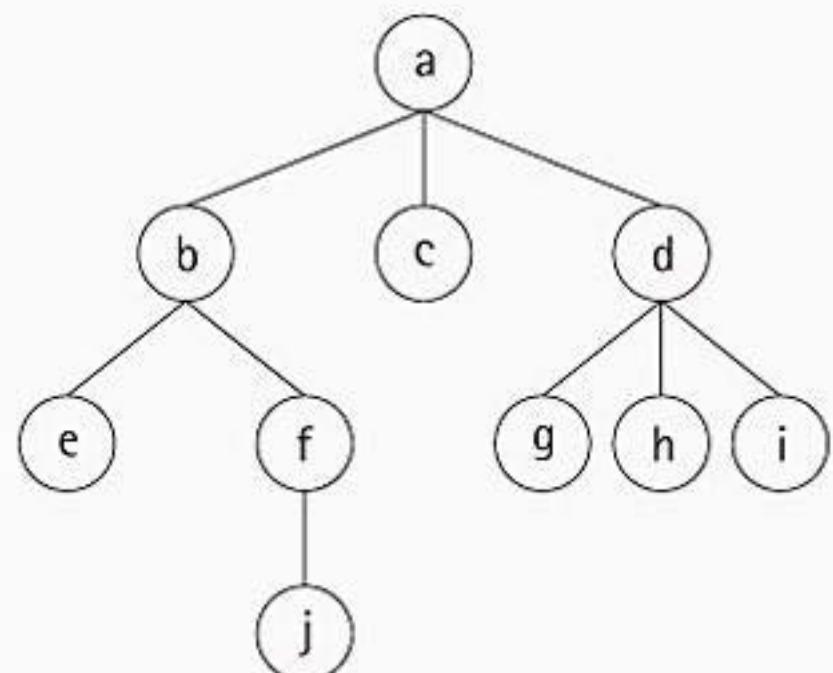


FIGURA 10.21 | Trajetória

Árvore como lista generalizada



Árvore binária

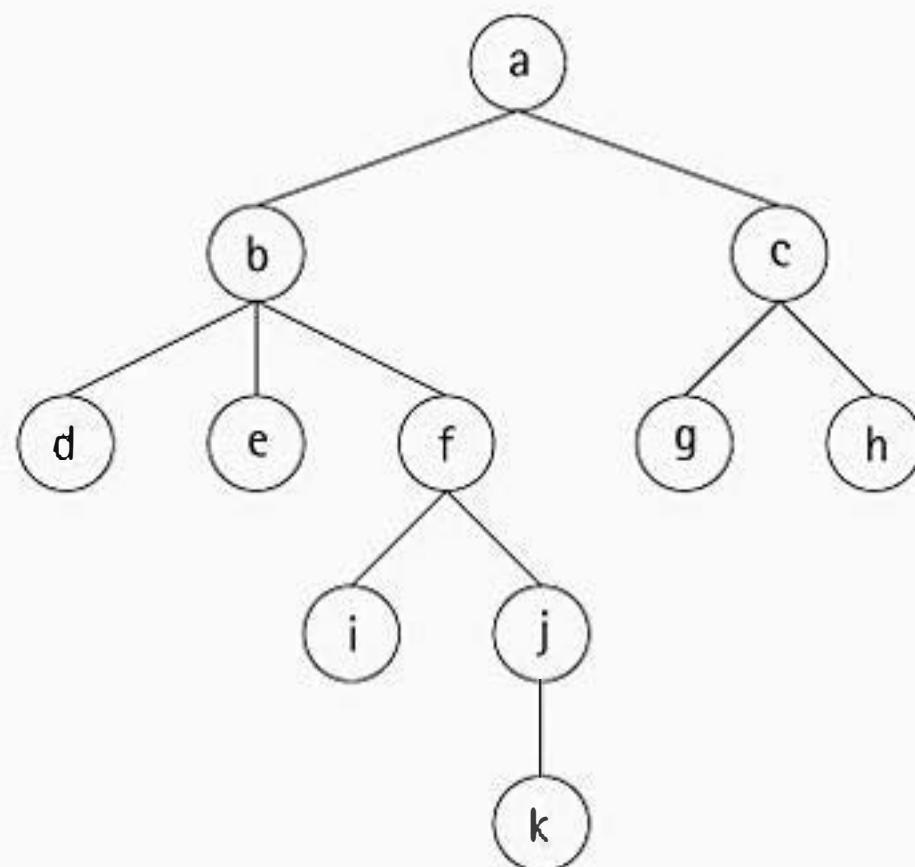


FIGURA 10.22 | Árvores

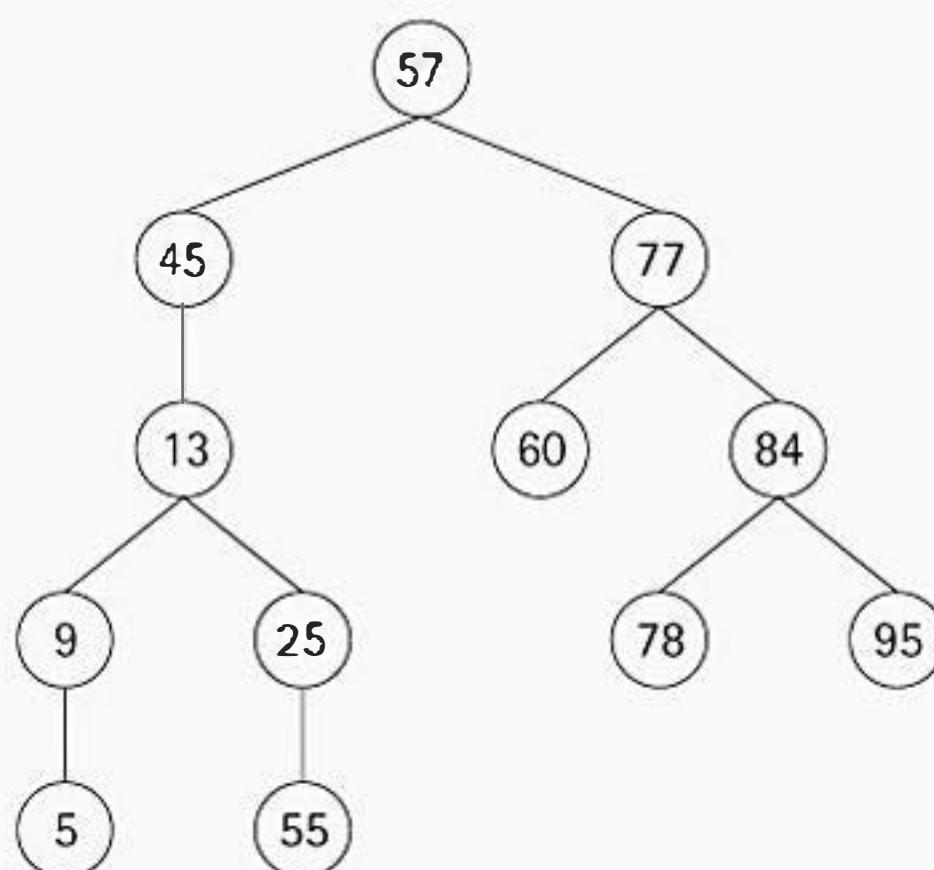


FIGURA 10.23 | Árvore binária

diretos (dois filhos). Nesse tipo de árvore também existe uma particularidade quanto à posição dos nós: os nós da direita sempre possuem valor superior ao do nó pai, e os nós da esquerda sempre possuem valor inferior ao do nó pai.

O algoritmo a seguir representa as variáveis que serão utilizadas para a manipulação da árvore — note que existe grande similaridade com os nós criados para manipulação das listas. O algoritmo tem a definição de um registro que possui as variáveis **valor**, **esq** e **dir**, a variável **apontador**, que será utilizada para fazer a referência a nós localizados à direita e à esquerda (da raiz ou do nó pai), e a variável **raiz**, que guardará o valor do nó raiz da árvore.

EXEMPLO 10.15: Pseudocódigo que representa uma árvore binária.

```
1. algoritmo BArvore
2.     tipo apontador: ^no_arvore
3.     tipo no_arvore = registro
4.         valor: inteiro
5.         esq: apontador
6.         dir: apontador
7.     fim
8.     var
9.         raiz: apontador
10.    Função inserir (arvore: no_arvore, novoNo: inteiro): no_arvore
11.    var
12.        apoio: no_arvore
13.    inicio
14.        Se (arvore = nulo) então
15.            apoio.valor ← novoNo
16.            retorno (apoio)
17.        Senão
18.            Se (novoNo < arvore.valor) então
19.                arvore^.esq ← inserir(arvore^.esq, novoNo)
20.            Senão
21.                arvore^.dir ← inserir(arvore^.dir, novoNo)
22.            Fim-se
23.        Fim-se
24.        retorno(arvore)
25.    Fim
26.
27.    Procedimento inserirNo (novoValor: inteiro)
28.    inicio
29.        raiz ← inserir(raiz, novoValor)
30.    fim
31.
32.    Procedimento exibir_esquerdo
33.        arv: no_arvore
34.        inicio
```

```

35.      arv ← raiz
36.      Se(arv <> nulo) então
37.          exibir_esquerdo(arv^.esq)
38.          mostre(arv^.valor)
39.      fim
40.
41. Procedimento exibir_direito
42.     arv: no_arvore
43.     inicio
44.         arv ← raiz
45.         inicio
46.             Se(arv <> nulo) então
47.                 exibir_direito(arv^.dir)
48.                 mostre(arv^.valor)
49.             fim
50.
51. Procedimento exibir_raiz( )
52.     inicio
53.         mostre("Raiz", raiz)
54.     fim

```

Os comentários serão feitos juntamente com os comentários do programa.

Para implementação de árvores binárias em Java, vamos utilizar os princípios anteriormente utilizados para a implementação de listas e filas. Criaremos uma classe **BIntNo**, que implementará o nó da árvore. A cada novo nó inserido na árvore, uma instância da classe **BIntNo** será criada, ou seja, um novo objeto nó.

EXEMPLO 10.16: Classe que representa o nó da árvore.

```

1. class BIntNo{
2.     int valor;
3.     BIntNo esq, dir;
4.
5.     BIntNo (int novoValor){
6.         valor = novoValor;
7.     }
8. }

```

No programa, assim como no algoritmo, podemos observar que o nó possui as variáveis **valor** do tipo inteiro, que guarda os elementos do nó, e as variáveis **esq** e **dir**, que são do tipo **BIntNo**. Na linha 5 temos o método construtor do objeto **BIntNo**, que recebe como parâmetro **novoValor**, que é o elemento do novo nó.

Da mesma forma que para o nó, criaremos uma classe para a árvore, que será instanciada toda vez que uma nova árvore for criada. Essa classe deverá conter os métodos que possibilitam a inclusão e exclusão de novos nós, conforme será visto a seguir.

EXEMPLO 10.17: Programa em Java que implementa uma árvore binária.

```

1. class BArvore{
2.     private BIntNo Raiz;
3.     private BIntNo inserir (BIntNo arvore, int novoNo) {
4.         if (arvore == null)
5.             return new BIntNo (novoNo);
6.         else if (novoNo < arvore.valor)
7.             arvore.esq = inserir (arvore.esq, novoNo);
8.         else
9.             arvore.dir = inserir (arvore.dir, novoNo);
10.        return arvore;
11.    }
12.    public void inserirNo (int novoValor) {
13.        Raiz = inserir (Raiz, novoValor);
14.    }
15.    private void exibir_esquerdo (BIntNo arv) {
16.        if (arv != null){
17.            exibir_esquerdo (arv.esq);
18.            System.out.println (arv.valor);
19.        }
20.    }
21.    private void exibir_direito (BIntNo arv) {
22.        if (arv != null){
23.            exibir_direito (arv.dir);
24.            System.out.println (arv.valor);
25.        }
26.    }
27.    public void exibir_raiz()
28.        {System.out.println("Raiz " + Raiz.valor);
29.    }
30.    public void exibirNo_esq (){
31.        exibir_esquerdo (Raiz);
32.    }
33.    public void exibirNo_dir (){
34.        exibir_direito (Raiz);
35.    }

```

A classe chamada `BArvore` possui como variável o nó `Raiz` do tipo `BIntNo`. `BIntNo` é um objeto que será instanciado toda vez que for inserido um novo nó na árvore. A classe `BArvore` possui os métodos `inserirNo`, para criar um novo nó na árvore, e `exibirNo_dir` e `exibirNo_esq` para mostrar todos os nós existentes. Vejamos cada um dos métodos/procedimentos da nossa árvore:

- `inserirNo` recebe como parâmetro um valor inteiro para a variável `novoValor`. Chama então o método `inserir`.
- `inserir` recebe como parâmetro um valor para `arvore`, que é uma variável do tipo `BIntNo`, e `novoNo`, que é do tipo inteiro. Esse método percorre

recursivamente a árvore a partir da raiz, buscando uma posição de referência nula para então inserir o novo elemento.

Vamos ver como isso funciona quando a árvore está vazia, isto é, quando Raiz tem valor nulo.

O método `inserirNo` chama o método `inserir`, passando como parâmetros: Raiz, que é nulo, e o valor do elemento. O método `inserirNo` (linha 12) recebe os valores, atribuindo Raiz à arvore e o valor do elemento a novoNo (que são as variáveis recebidas como parâmetro no método `inserir`).

Quando é feita a verificação na linha 4, como arvore tem valor nulo, o método retorna um novo nó, que passará a ser o nó raiz. Quando a árvore já possui mais de um elemento, o método `inserirNo` chama o método `inserir`, passando os parâmetros: Raiz e o valor do elemento. O método `inserir` verifica se o valor a ser inserido é maior ou menor que o nó (linhas 4 a 10). Sendo menor, insere-o à esquerda, caso contrário insere-o à direita. A recursividade é acionada nas linhas 7 (para nós à esquerda) e 9 (para nós à direita). Enquanto a referência de um nó (`arvore.esq` ou `arvore.dir`) não for um valor nulo, o método `inserir` continuará a ser chamado. Quando isso ocorrer, o método retornará um novo nó que passará a fazer parte da árvore. Dessa forma, uma referência leva a outra até que se encontre um nó vazio (nulo), que possibilite a inserção de um novo nó.

NOTA:

A recursividade se dá, nesse caso, com uma chamada interna do próprio método no qual ela está inserida.

Deve-se observar, também, o que chamamos de visibilidade de um método. Os métodos declarados como `public` podem ser chamados ou invocados por outros procedimentos em execução, enquanto os métodos declarados como `private`, somente dentro da própria classe (vide Anexo).

- Os métodos `exibirNo_esq` (linha 29) e `exibirNo_dir` (linha 32) usam o mesmo princípio do método `inserirNo`. Eles não recebem parâmetros e chamam os métodos `exibir_esquerdo` ou `exibir_direito`, enviando como parâmetro o nó Raiz.
- Os métodos `exibir_direito` e `exibir_esquerdo` recebem como parâmetro um valor para `arv`, do tipo `BIntNo`, que é passado pelo método `exibirNo_esq` ou `exibirNo_dir`. Por meio de chamadas recursivas, buscam os nós à esquerda ou à direita da árvore. Encontrando uma referência nula, isto é, se `arv` não for diferente de nulo (linhas 16 e 21), mostra-se o valor do nó (linhas 18 e 23). Esse processo garante o percurso na árvore ao recuperar e imprimir uma saída sempre na ordem ascendente de valor.

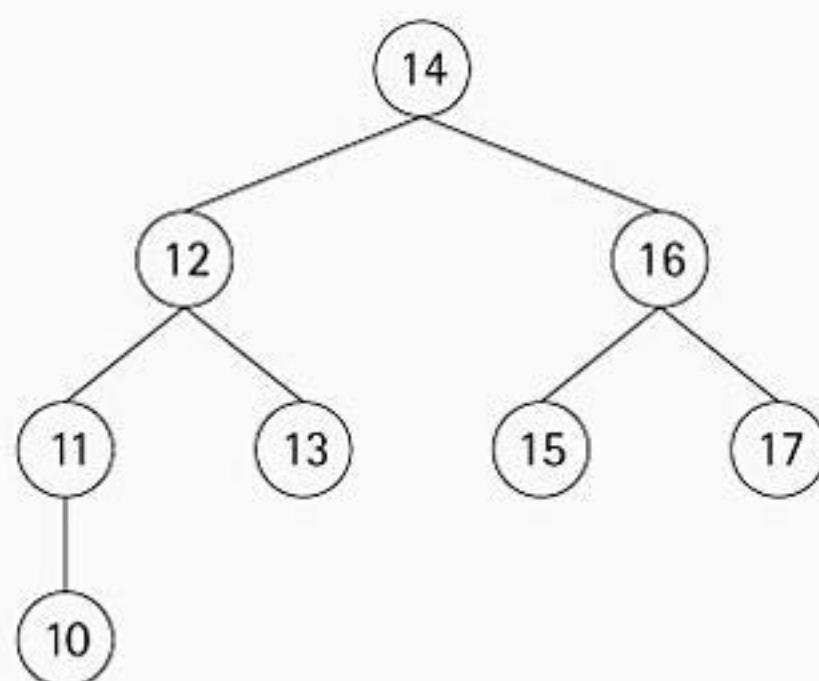
O exemplo a seguir faz a inclusão dos nós em uma árvore passando os valores de forma aleatória, exibe o conteúdo, insere novos nós (linhas 11 e 12) e exibe o resultado novamente. Como o programa, constrói a árvore obedecendo à relação: nós à esquerda possuem valor inferior ao nó pai e nós à direita possuem valor superior. A exibição do resultado (método `exibirNo`) produz uma saída ordenada dos elementos por valor e ascendente. Lembre-se de que no exemplo é instanciado um objeto `BArvore` com o nome `arvore1`, cujo código apresentamos anteriormente.

EXEMPLO 10.18: Classe que utiliza a árvore implementada no Exemplo 10.17.

```

1. class Exemplo106{
2.     public static void main(String[] args){
3.         BArvore arvore1 = new BArvore ();
4.         arvore1.inserirNo (14);
5.         arvore1.inserirNo (16);
6.         arvore1.inserirNo (12);
7.         arvore1.inserirNo (11);
8.         arvore1.inserirNo (17);
9.         arvore1.inserirNo (15);
10.        arvore1.exibirNo ();
11.        arvore1.inserirNo (10);
12.        arvore1.inserirNo (13);
13.        System.out.println ("");
14.        arvore1.exibirNo ();
15.    }
16. }
```

A figura abaixo representa a árvore do Exemplo 10.18 após a inserção dos nós.



|FIGURA 10.24| Representação da árvore após a inserção dos nós

Fizemos a inclusão e exibição dos elementos em uma árvore. Vamos agora implementar a exclusão, lembrando que a exclusão de um nó é um processo um pouco mais complexo, uma vez que as referências ao nó excluído e a seus filhos precisam ser devidamente ajustadas.

LEMBRE-SE: Vamos apresentar apenas o trecho correspondente ao processo de exclusão, mas ele é continuação do algoritmo BArvore, e o método excluirNo é continuação da classe BArvore.

EXEMPLO 10.19: Pseudocódigo para representar o procedimento de exclusão de nós com árvores binárias.

```

1. Procedimento excluirNo(item: inteiro)
2. var
3.     tempNo: no_arvore
4.     pai: no_arvore
5.     filho: no_arvore
6.     temp: no_arvore
7. inicio
8.     tempNo ← raiz
9.     pai ← nulo
10.    filho ← raiz
11.    Enquanto(tempNo <> nulo .e. tempNo^.valor <> item) faça
12.        pai ← tempNo
13.        Se(item < tempNo^.valor) então
14.            tempNo ← tempNo^.esq
15.        Senão
16.            tempNo ← tempNo^.dir
17.        Fim-se
18.        Se(tempNo = nulo) então
19.            Mostre("Item não localizado")
20.        Fim-se
21.        Se(pai = nulo) então
22.            Se(tempNo^.dir = nulo)
23.                raiz ← tempNo^.esq
24.            Fim-se
25.            Se(tempNo.esq = nulo)
26.                raiz ← tempNo^.dir
27.            Fim-se
28.        Senão
29.            temp ← tempNo
30.            filho ← tempNo^.esq
31.            Enquanto(filho.dir <> nulo) faça
32.                temp ← filho
33.                filho ← filho^.dir
34.            Fim-enquanto
35.            Se (filho < > tempNo^.esq) então
36.                temp^.dir ← filho.^esq
37.                filo.^esq ← raiz.^esq
38.            Fim-se
39.            filho.^dir ← raiz.^dir
40.            raiz ← filho
41.        Fim-se

```

```

42.      Se (tempNo.^dir = nulo) então
43.          Se(pai.^esq = tempNo.^esq
44.              pai.^esq ← tempNo.^esq
45.          Senão
46.              pai.^dir ← tempNo.^esq
47.          Fim-se
48.      Senão
49.          Se(tempNo.^esq = tempNo) então
50.              Se(pai.^esq = tempNo) então
51.                  pai.^esq ← tempNo.^dir
52.              Senão
53.                  pai.^dir ← tempNo.^dir
54.              fim-se
55.          Senão
56.              temp ← tempNo
57.              filho ← tempNo.^esq
58.              Enquanto(filho.dir <> nulo) faça
59.                  temp ← filho
60.                  filho ← filho.^dir
61.              Fim-enquanto
62.              Se(filho <> tempNo.^esq) então
63.                  temp.^dir ← filho.^esq
64.                  filho.^esq ← tempNo.esq
65.              Fim-se
66.              filho.^dir ← tempNo.^dir
67.              Se(pai.^esq = tempNo) então
68.                  pai.^esq ← filho
69.              Senão
70.                  pai.^dir ← filho
71.              fim-se
72.          Fim-se
73.      fim

```

A seguir apresentamos o método `excluirNo`, que implementa o processo de exclusão.

```

1. public void excluirNo (int item){
2.     try{
3.         BIntNo tempNo = Raiz, pai = null, filho = Raiz, temp;
4.         while (tempNo != null && tempNo.valor != item){
5.             pai = tempNo;
6.             if (item < tempNo.valor)
7.                 tempNo = tempNo.esq;
8.             else
9.                 tempNo = tempNo.dir;
10.        }
11.        if (tempNo == null)

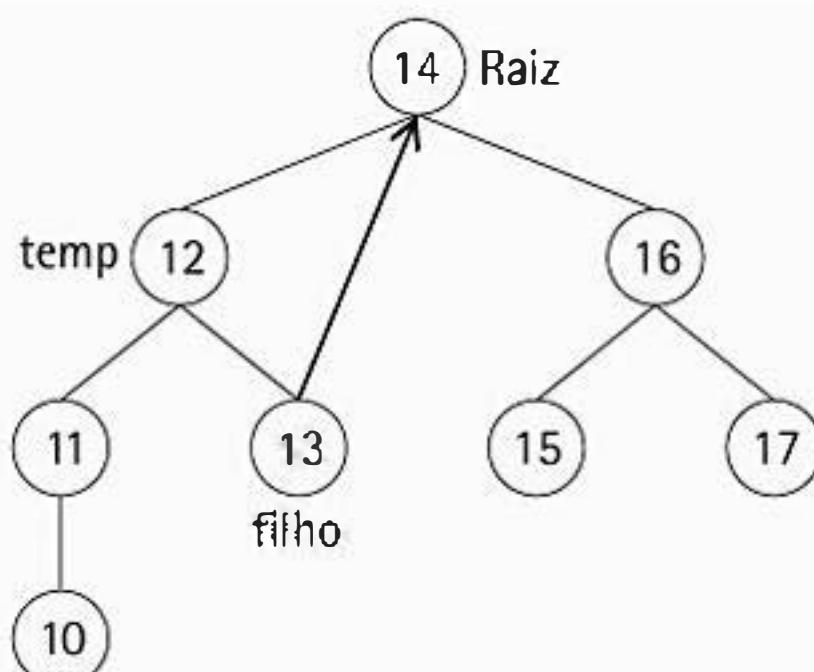
```

```
12.         System.out.println ("Item nao localizado.");
13.     if (pai == null){
14.         if (tempNo.dir == null)
15.             Raiz = tempNo.esq;
16.         else if (tempNo.esq == null)
17.             Raiz = tempNo.dir;
18.         else{
19.             for (temp = tempNo, filho = tempNo.esq;
20.                  filho.dir != null; temp = filho, filho = filho.dir);
21.             if (filho != tempNo.esq){
22.                 temp.dir = filho.esq;
23.                 filho.esq = Raiz.esq;
24.             }
25.             filho.dir = Raiz.dir;
26.             Raiz = filho;
27.         }
28.     } else if (tempNo.dir == null){
29.         if (pai.esq == tempNo)
30.             pai.esq = tempNo.esq;
31.         else
32.             pai.dir = tempNo.esq;
33.     } else if (tempNo.esq == null){
34.         if (pai.esq == tempNo)
35.             pai.esq = tempNo.dir;
36.         else
37.             pai.dir = tempNo.dir;
38.     } else {
39.         for (temp = tempNo, filho = tempNo.esq;
40.              filho.dir != null; temp = filho, filho = filho.dir);
41.         if (filho != tempNo.esq){
42.             temp.dir = filho.esq;
43.             filho.esq = tempNo.esq;
44.         }
45.         filho.dir = tempNo.dir;
46.         if (pai.esq == tempNo)
47.             pai.esq = filho;
48.         else
49.             pai.dir = filho;
50.     }
51. } catch (NullPointerException erro) {
52.     //Item nao encontrado
53. }
54. }
55. }
```

- O método excluirNo recebe como parâmetro o valor do elemento a ser excluído.

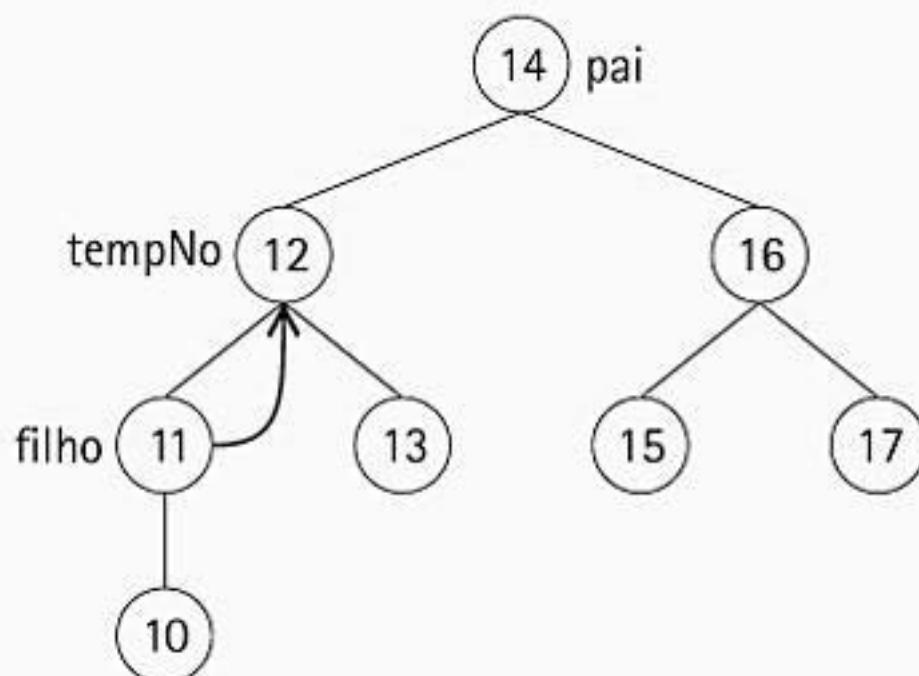
O trecho das linhas 4 a 10 faz a busca nos diversos nós da árvore, comparando o valor fornecido (`item`) com o valor do nó. Posteriormente, são tratadas as diversas possibilidades, conforme segue:

- linha 11 — se `tempNo` for nulo, significa que a árvore foi percorrida totalmente e o valor não foi encontrado;
- linha 13 — se `pai` for nulo, significa que o nó pesquisado encontra-se na própria raiz e a condição da linha 4 não foi satisfeita, ou seja, o valor de `tempNo` é igual a `item`, que é o valor pesquisado;
- linha 14 — se `tempNo.dir` for nulo, significa que o nó não possui filho à direita, bastando atribuir a `Raiz` o nó à esquerda, dado por `tempNo.esq`;
- linha 16 — se `tempNo.esq` for nulo, significa que o nó não possui filho à esquerda, bastando atribuir a `Raiz` o nó à direita, dado por `tempNo.dir`;
- linha 20 — se o nó tiver filhos à esquerda e à direita, é executado o laço `for` à linha 19, que busca pelo elemento mais à direita do ramo esquerdo da árvore (maior valor em relação à raiz que será excluída). Esse elemento é trocado pela raiz e as referências ajustadas. Isso pode ser melhor visualizado pelo esquema a seguir, que usa os valores do Exemplo 10.18.



|FIGURA 10.25| Representação da exclusão de um elemento da árvore

- linha 28 — se `pai` não for nulo — condição senão do `if` à linha 38 —, significa que o nó a ser excluído não é o nó raiz;
- linha 29 — se o nó a ser excluído não possuir filho à direita, a referência do nó pai é trocada pela do nó a ser excluído;
- linha 33 — se o nó a ser excluído não possuir filho à esquerda, a referência do nó pai é trocada pela do nó a ser excluído;
- linha 38 — se o nó a ser excluído possuir filhos à esquerda e à direita, é utilizado procedimento semelhante à exclusão do nó raiz, porém, nesse caso, devem ser ajustadas as referências do pai do nó a ser excluído. Usando os valores do Exemplo 10.18, pode-se visualizar esquematicamente, pela figura a seguir, como o rearranjo da árvore é realizado.



| FIGURA 10.26 | Representação da exclusão de um elemento da árvore

É importante observar que nesse exemplo o ramo esquerdo do nó a ser excluído não possui nó à direita. A busca pelo nó mais à direita (linha 39) retorna como filho o nó à esquerda de `tempNo`, pela própria condição do laço. De fato, esse nó representa o maior valor dos elementos do ramo esquerdo do nó a ser excluído.

Para o entendimento do funcionamento do processo é importante simular a entrada de valores e a construção da respectiva árvore graficamente. A modificação na ordem e na quantidade dos valores de entrada e dos nós excluídos dará uma idéia mais clara do comportamento assumido para cada caso.

O exemplo a seguir implementa a entrada de valores, a exclusão de um nó e a exibição dos nós da árvore antes e depois da exclusão.

EXEMPLO 10.20: Programa que utiliza a árvore binária criada nos exemplos 10.18 e 10.19.

```

1. class Exemplo107{
2.     public static void main(String[] args){
3.         BArvore2 arvore1 = new BArvore2 ();
4.         /* inserção de nos na arvore */
5.         arvore1.inserirNo (14);
6.         arvore1.inserirNo (16);
7.         arvore1.inserirNo (12);
8.         arvore1.inserirNo (11);
9.         arvore1.inserirNo (17);
10.        arvore1.inserirNo (15);
11.        arvore1.inserirNo (10);
12.        arvore1.inserirNo (13);
13.
14.        /* exibição dos nós da árvore */
15.        arvore1.exibir_raiz();
16.        System.out.println("No esquerdo");
17.        arvore1.exibirNo_esq();
18.        System.out.println("No direito");
19.        arvore1.exibirNo_dir();
20.    }
  
```

```
21.     /* exclusao de um nó */
22.     System.out.println("valor excluido 50");
23.     arvore1.excluirNo(12);
24.
25.     /* exibicao da árvore após a exclusão */
26.     System.out.println("exibicao da arvore apos a exclusao");
27.     arvore1.exibir_raiz();
28.     System.out.println("No esquerdo");
29.     arvore1.exibirNo_esq();
30.     System.out.println("No direito");
31.     arvore1.exibirNo_dir();
32. }
33. }
```

10.7 EXERCÍCIOS PARA FIXAÇÃO

1. Leia cem números e construa uma pilha apenas com os números pares.
2. Utilizando a estrutura de uma pilha, leia uma palavra ou frase e a exiba na ordem inversa.
3. Considere que o usuário fará a entrada de valores (números e operadores) que irão representar uma expressão matemática. A expressão poderá conter parênteses para isolar as operações. O programa deverá verificar se todos os parênteses estão fechados. Para isso represente o empilhamento dos '(' e ')'; cada vez que um par for formado deverá ser desempilhado, e se sobrar algum elemento na pilha significará que a expressão não está correta.
4. Descreva a saída da seguinte seqüência sobre a pilha:
empilha(5), empilha(3), desempilha(), empilha(2), empilha(8), desempilha(),
desempilha(), empilha(9), empilha(1), desempilha(), empilha(7), empilha(6),
desempilha(), desempilha(), empilha(4), desempilha(), desempilha().
5. A conversão de um valor decimal para o seu correspondente em binário se dá pelas sucessivas divisões dele por 2 até que o quociente seja 0. O representante binário desse número será composto por todos os restos, mas na ordem inversa à que foram calculados. Elabore um algoritmo e um programa capazes de resolver essa questão utilizando o conceito de pilhas. O número deverá ser fornecido pelo usuário.
6. Cite e justifique pelo menos três atividades realizadas em computadores que requeiram o uso de pilhas.
7. Escreva uma fila com implementação por meio de arranjos, cuja capacidade seja para quatro elementos. Na fila deverão ser enfileirados apenas os números ímpares.
8. Considerando a solução do Exercício 6, faça o teste da seqüência abaixo indicando o valor da saída e o resultado da fila.

Instruções: na coluna ‘saída’ mostre uma das mensagens: Valor enfileirado, Valor não atende a condição, Erro! Fila cheia, Erro! Fila vazia, Verdadeiro, Falso; na coluna ‘resultado da fila’ mostre todos os elementos da fila na passagem.

Operação	Saída	Resultado da fila
desenfileirar()		
enfileirar(5)		
enfileirar(8)		
desenfileirar()		
enfileirar(7)		
enfileirar(4)		
enfileirar(1)		
enfileirar(3)		
desenfileirar()		
desenfileirar()		
desenfileirar()		
vazio()		
enfileirar(9)		
enfileirar(7)		
tamanho()		
enfileirar(13)		
enfileirar(15)		
desenfileirar()		

9. Cite e justifique pelo menos três atividades realizadas em computadores que requeiram o uso de filas.
10. Construa um algoritmo/programa que administre as filas de reservas de filmes em uma videolocadora, levando em conta que para cada filme existem sete filas — uma para cada dia da semana — e é o usuário quem determina qual é o dia da semana de sua preferência para alugar o filme. O cliente é informado da disponibilidade da fita e quando é confirmada a sua locação ele deve sair da fila. O número de cópias de cada fita não deverá ser considerado nessa solução, devendo ser considerada uma fita de cada filme.
11. Crie uma lista com os itens que você precisa comprar, podendo ser material de escritório ou produtos para sua casa. Faça o pseudocódigo e a implementação em Java, de forma que, à medida que os itens forem adquiridos, possam ser removidos da lista.

12. Implemente o Exercício 6 utilizando o conceito de lista simples.
13. Implemente o Exercício 2 utilizando o conceito de listas duplamente encadeadas.
14. Crie uma árvore para armazenar os elementos da seguinte expressão matemática: $A + B * C - D / E$. Cada letra ou símbolo matemático deverá ocupar a posição de um elemento. O arranjo dos elementos deverá seguir uma ordem que gere uma saída representando a ordem em que a expressão deve ser executada, considerando a precedência dos operadores.
15. Crie uma árvore que receba palavras. O programa/algoritmo deve contar as palavras, que deverão ser inseridas pelo usuário por meio do teclado. Deve ser criado um menu com as opções para: inserir uma nova palavra, excluir uma palavra fornecida pelo usuário e exibir a árvore.
16. Considere um labirinto qualquer, representado por uma matriz 20×20 e crie um programa/algoritmo para:
 - a) encontrar um caminho entre o ponto de entrada e saída;
 - b) armazenar as posições já percorridas;
 - c) ser capaz de retornar em um caminho que não tem saída.

Observações: o usuário deve escolher o caminho a ser percorrido via console; para isso defina padrões de movimentação, por exemplo: direita, esquerda, acima e abaixo.

ANEXO: CONCEITOS SOBRE A LINGUAGEM JAVA

Java é uma linguagem de programação orientada a objetos desenvolvida pela Sun Microsystems para ser portável entre diferentes plataformas.

ALGUMAS CARACTERÍSTICAS

- É orientada a objetos.
- É portável, isto é, seus programas podem ser utilizados em diferentes plataformas.
- Seu código de programação é interpretado, o que garante maior velocidade de desenvolvimento e portabilidade.
- Permite que o programa execute mais de um thread (linha de execução).

A linguagem de programação Java possui um ambiente de desenvolvimento e um ambiente de aplicativos. Ela resulta da busca de uma linguagem com as características do C++ aliadas à segurança do Smalltalk.

RECURSOS

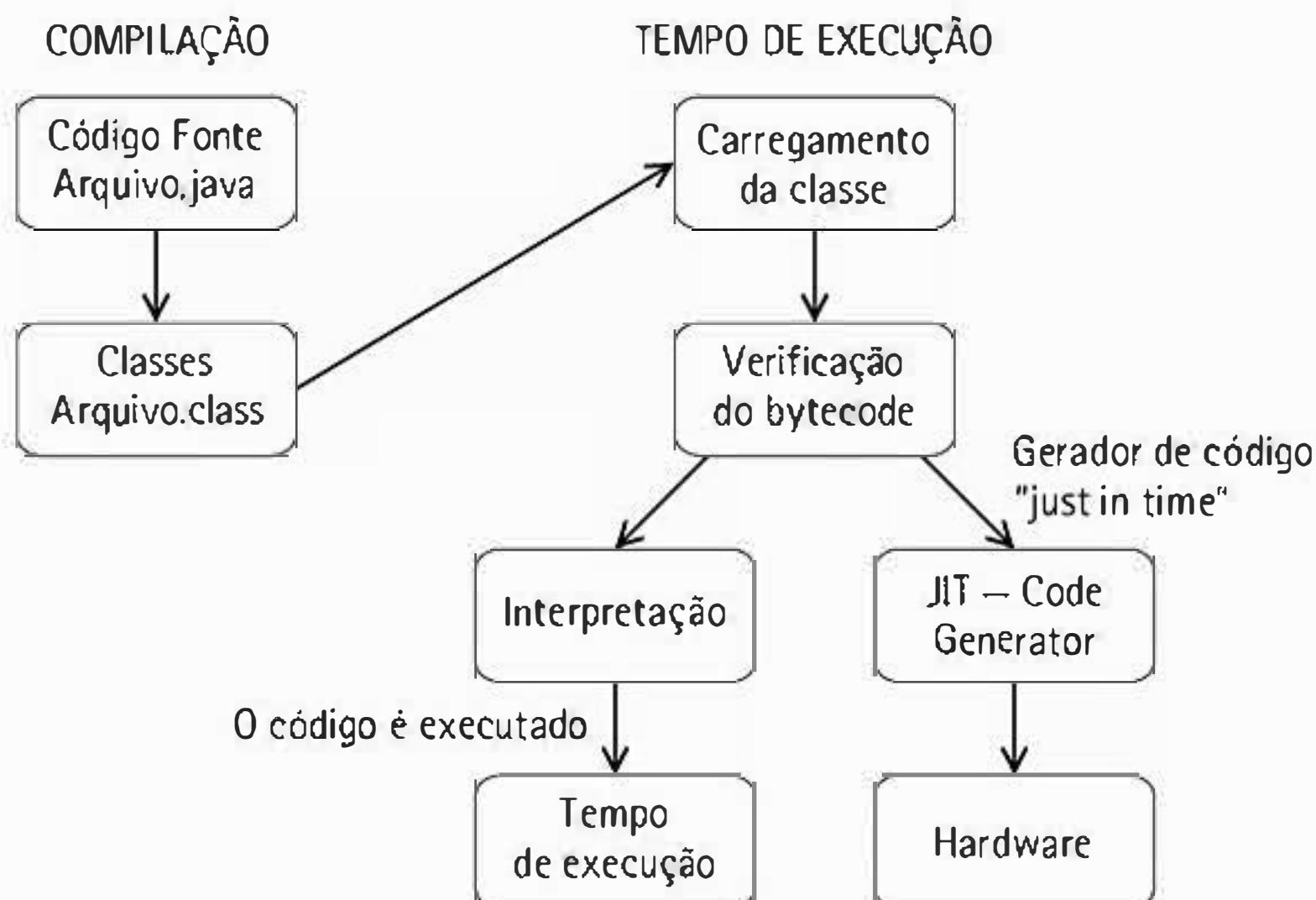
O JVM — Java Virtual Machine — cria uma máquina imaginária, implementada por meio da emulação de um software executado em uma máquina real. O código do JVM está armazenado em um arquivo .class.

O JVM fornece as especificações da plataforma de hardware, o que permite a independência de plataforma, uma vez que o programa é compilado em uma máquina imaginária. Essas especificações consistem em instruções e registros da linguagem de máquina, tamanho do cache, velocidade do processador etc.

COLETA DE LIXO

A garbage collection varre as memórias cujo ponteiro possui número de referência 0. Ocorre automaticamente durante o tempo de vida de um programa.

AMBIENTE JAVA DE COMPILAÇÃO



|FIGURA 1| Ambiente Java de compilação

DURANTE O PROCESSO DE COMPILAÇÃO

Os arquivos são compilados à medida que são convertidos para bytecodes. Em tempo de execução, os bytecodes são carregados, verificados e executados. Durante a execução, são feitas as chamadas ao hardware subjacente.

O carregamento de classe é responsável pela carga de todas as classes necessárias à execução do programa.

O interpretador verifica se foi gerado algum código ilegal, com a finalidade de verificar se o bytecode adere às especificações JVM e não viola a integridade e a segurança do sistema.

Na geração de código, os bytecodes são compilados no código de máquina nativo antes da execução, isto é, o código é interpretado para a plataforma em questão.

Para desenvolver programas, será necessário utilizar o Java Development Kit (JDK), um ambiente de desenvolvimento que, além das ferramentas de desenvolvimento, contém todos os elementos da plataforma Java. É possível adquirir o JDK gratuitamente no site da Sun: www.sun.com.br.

ARQUIVOS DO JDK

Quando se instala o JDK, é criada a seguinte estrutura de diretório:

Diretório	Conteúdo
.hotjava	Configuração do applet viewer para uso do JDK.
bin	Arquivos binários (executáveis e bibliotecas) do JDK.
demo	Exemplos de aplicações e applets.
docs	Documentação da API.
include	Arquivos utilizados para integração com código nativo.
lib	Classes da API.
src	Código fonte das classes. Somente será criado se for especificado como opção na instalação do JDK. Os arquivos desse diretório podem ser utilizados para depuração.

ALGUMAS DEFINIÇÕES:

Modificadores — têm a função de modificar a atribuição de classes, métodos e variáveis-membro. São utilizados como prefixos.

Construtores — são utilizados para inicializar um objeto após a sua criação. Um construtor tem o mesmo nome da classe em que reside e não tem tipo de retorno, pois o tipo de retorno é implícito e é do mesmo tipo da classe em que reside.

Métodos — são os conjuntos de instruções executáveis de uma classe.

Código fonte — são os arquivos que contêm as instruções escritas na linguagem de programação escolhida, no caso, Java. O código fonte pode ser escrito em um editor de textos como o Edit do DOS ou o Notepad do Windows. Os arquivos devem ser salvos com um nome (que deve ser idêntico ao nome da classe), com um ponto (.) e com a extensão, que deve ser .java. Por exemplo: teste.java, media.java, cadastro.java etc.

```

1. // Exemplo de programa em Java
2. public class Olá
3. {public static void main (String args[])
4.     { System.out.println("Olá para todos!");
5.     }
6. }
```

ENTENDENDO AS INSTRUÇÕES DO CÓDIGO FONTE:

1. // inicia um comentário, utilizado para incluir explicações no meio do código.
2. A palavra **public** é um especificador de acesso que permite ao programador controlar a visibilidade de uma classe ou método. **class Olá** é a identificação do nome da classe.

3. `static void main` — o `main` é o principal método. O Java procura esse método para interpretar uma classe. O `void` é um modificador utilizado quando não é necessário retorno. O `main` é declarado como `static`, pois não é preciso fazer referência a uma instância, uma vez que ele é chamado pelo interpretador antes que quaisquer instâncias sejam criadas, isto é, o `main` não irá requerer a chamada de uma instância dessa classe. O `String args[]` determina a criação de um vetor chamado `args`, com elementos do tipo `String` utilizados para a entrada de parâmetros.
5. `System` — nome da classe; `out` — nome do objeto da classe `System`, responsável pela saída; `println` — método utilizado para exibir mensagens na tela do computador.
6. Os blocos de programa são delimitados por chaves. Esses blocos devem ser abertos `{` e fechados `}`.

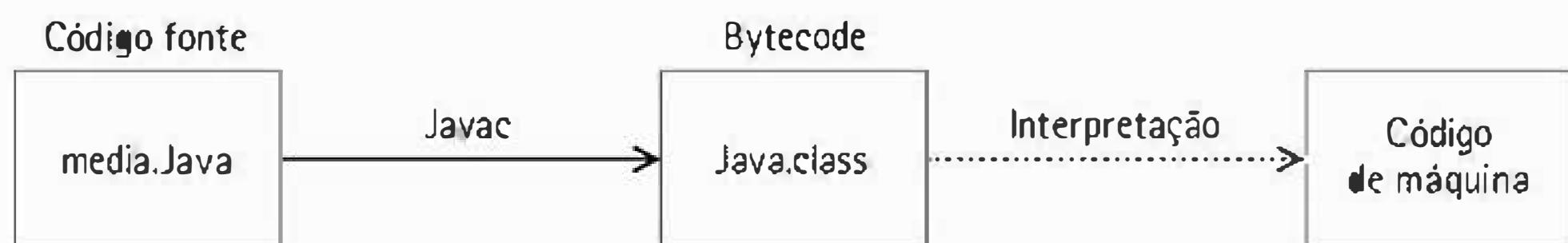
NOTA:

Esse arquivo deve ser gravado com o nome ola.java, pois deve levar o mesmo nome da classe.

COMPILADOR JAVAC

O código fonte contém instruções de alto nível, isto é, instruções em uma linguagem que o homem é capaz de compreender, mas a máquina não. Por isso, o código fonte precisa ser compilado. O arquivo responsável pelo processo de compilação em Java é o Java compiler, chamado `javac`.

O `javac` gera, a partir do código fonte, um arquivo de bytecodes. Esse arquivo terá o mesmo nome do arquivo fonte, mas com a extensão `.class`. Por exemplo: `teste.class`, `media.class`, `cadastro.class` etc.



[FIGURA 2] Diagrama de representação do processo de compilação

Para compilar um arquivo, basta digitar `javac` e o nome do arquivo. Por exemplo, no prompt do DOS, conforme o modelo:

C:\> javac ola.java

Existem algumas opções que podem ser utilizadas para compilar o programa. Para isso, utiliza-se a sintaxe `javac [opções] arquivo [arquivo...]`. Essas opções são relacionadas a seguir:

`javac [opções] arquivo [arquivo...]`

OPÇÕES:

-classpath *path{;...}* — localização das classes já definidas. Sobrepõe a variável de ambiente CLASSPATH.

-d *dir* — especifica o diretório raiz onde as classes compiladas serão armazenadas.

-deprecation — ativa as mensagens de advertência que indicam os membros ou as classes que estão em desuso.

-g — cria tabelas de debugging que serão usadas pelo debugger. Contém informações como variáveis locais e números de linha. A ação default do compilador é somente gerar números de linhas. A opção -g:nodetbug suprime a geração dos números de linha.

-nowarn — desativa as mensagens de advertência que informam problemas potenciais com o código fonte.

-verbose — mostra informações adicionais sobre a compilação. Contrasta com **-nowarn**.

-depend — efetua a compilação de todos os arquivos que dependem do código fonte que está sendo compilado. Sem essa opção, são compilados apenas os arquivos cujas classes são invocadas no arquivo que está sendo compilado.

-O — otimiza o código compilado para gerar programas mais rápidos. Na versão 1.1, inclui o código de todos os métodos, em vez de simplesmente invocá-los (call); aumenta o tamanho dos arquivos compilados.

-J *opção* — passa a string *opção* como argumento para o interpretador que irá executar esse código. A string opção não pode conter espaços em branco. O argumento **-J** pode ser utilizado múltiplas vezes.

Exemplo:

```
javac -classpath .;C:\classes Ola.java
javac Ola.java Teste.java
```

PARA INTERPRETAR O PROGRAMA

Para executar (utilizar) o programa, basta digitar, no prompt do DOS, a palavra java seguida do nome do programa. Por exemplo:

```
c:\> java Ola
```

PALAVRAS RESERVADAS

No Capítulo 3, foi abordado o assunto identificadores (nomes) de variáveis, sobre o qual foi dito que não se devem utilizar palavras reservadas à linguagem de programação. A seguir são apresentadas as palavras reservadas à linguagem Java.

abstract	default	goto	null	synchronized
boolean	do	if	package	this

break	double	implements	private	throw
byte	else	import	protected	throws
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	volatile
continue	for	new	switch	while

Essas palavras não podem ser usadas como nomes de variáveis, métodos, interfaces ou classes.

VARIÁVEIS

A declaração de variáveis em Java pode ser feita em qualquer parte do programa, e a variável pode ser declarada e inicializada. Para relembrar os tipos de dados em Java, consulte o Capítulo 3.

Exemplos:

```
byte idade;
short a1;
int i = 100;
long l = 500L; /* L: identificador long */
float v1,v2=4.578f; /* f:identificador float */
double d = 5,02e19; /* notação científica */
char h = 'h';
boolean resposta = true;
/* String são objetos, mas podem ser inicializados como tipos
primitivos */
String cadeia = "String em Java não é tipo primitivo";
```

As variáveis podem ser declaradas dentro de uma classe. São as chamadas **variáveis-membro**, que podem ser acessadas por qualquer conjunto de instruções da classe. Já as variáveis declaradas dentro de um dos métodos da classe são chamadas de **variáveis locais**.

MODIFICADORES DE VARIÁVEIS

public — a variável é visível por todos.

private — a variável só é visível pela classe.

protected — a variável só é visível para a classe onde foi criada e para suas herdeiras.

private protected — a variável é visível somente para as subclasses.

static — a variável será chamada pelo nome da classe e não por um objeto dela.

final — a variável não pode ter seu valor alterado, pois é uma constante.

COMENTÁRIOS

Às vezes torna-se importante a escrita de textos no meio do código do programa para explicar algo. Isso é denominado **comentário** e é um recurso muito utilizado para documentar partes do código. Em Java, os comentários podem ser feitos de três maneiras:

`//` — para uma linha de comentário, por exemplo: `// comentário`

`/*` — para um trecho de comentários. O trecho deve ser delimitado, por exemplo: `/* trecho */`

`/**` — para documentação. O trecho deve ser delimitado, por exemplo: `/** trecho */`

MODIFICADORES DE CLASSES

`public` — pode ser instanciada e utilizada livremente, mesmo na classe interna.

`protected` — pode ser utilizada apenas no mesmo package ou em subclasses da classe externa.

`private` — pode ser utilizada apenas na classe externa.

`abstract` — não pode ser instanciada. Esse modificador é utilizado para definir apenas superclasses genéricas.

`final` — não permite definição de subclasses.

MODIFICADORES DE MÉTODOS E VARIÁVEIS-MEMBRO

`public` — pode ser chamado livremente e as variáveis-membro podem ser utilizadas livremente.

`protected` — os métodos e as variáveis-membro podem ser chamados apenas no mesmo package e em subclasses; as variáveis-membro seguem o mesmo padrão.

`private` — os métodos e as variáveis-membro podem ser chamados apenas na classe.

`abstract` — não podem ser definidos nem métodos nem variáveis-membro.

`final` — os métodos não podem ser sobreescritos; é utilizado para declarar **constantes**.

`static` — tanto os métodos como as variáveis-membro podem ser invocados a partir do nome da classe.

`native` — é utilizado para a definição de métodos em código nativo. Não se aplica a variáveis-membro.

`transient` — não se utiliza em métodos e não há persistência em variáveis-membro.

`synchronized` — para métodos, não permite acesso simultâneo a programas multithread. Não se aplica a variáveis-membro.

`volatile` — não é aplicável à definição de métodos e em variáveis-membro não permite cache.

NOTA:

A ordem dos modificadores não importa.

EXCEÇÕES

O tratamento de exceções é realizado por um método que o Java possui para detecção de erros. O tratamento de exceções é também conhecido como tratamento de erros, pois os erros são capturados por esse recurso, que gera um alerta para o usuário.

As exceções ou erros são derivados da superclasse `Throwable`, conforme pode ser observado na Figura 3.

Descrição de algumas classes

`Exception` — são exceções genéricas que devem ser tratadas.

`Error` — são erros graves que normalmente não podem ser tratados.

`RuntimeException` — são as exceções geradas por erros de programação.

`NullPointerException` — é uma exceção associada à tentativa de acesso a propriedades ou métodos de referências com valor `null`.

`ArithmaticException` — é uma exceção associada à tentativa de utilização incorreta de operadores aritméticos como, por exemplo, a divisão por zero.

`IndexOutOfBoundsException` — é uma exceção associada a índices de vetores ou strings.

`ArrayStoreException` — é uma exceção gerada pela tentativa de atribuir a um elemento de um array um objeto incompatível com o tipo declarado do array.

A captura das exceções deve ser feita da seguinte forma:

```
try {
    // código que PODE gerar uma exceção
} catch(Exception e) { // captura da exceção
    // código executado no tratamento da exceção
}
```

Sendo que o código que pode gerar uma exceção deve ser descrito no bloco `try`. Essas exceções, por sua vez, são capturadas no bloco `catch()`, que deve especificar os argumentos (exceções a serem tratadas).

Passagem da exceção pelo método:

```
void metodo1() throws IOException {
    // código que pode gerar uma exceção IOException
}
```

Lançamento de exceção:

```
void metodo2() throws IOException {
    // testa condição de exceção
    if(excep) then throw(new IOException());
}
```

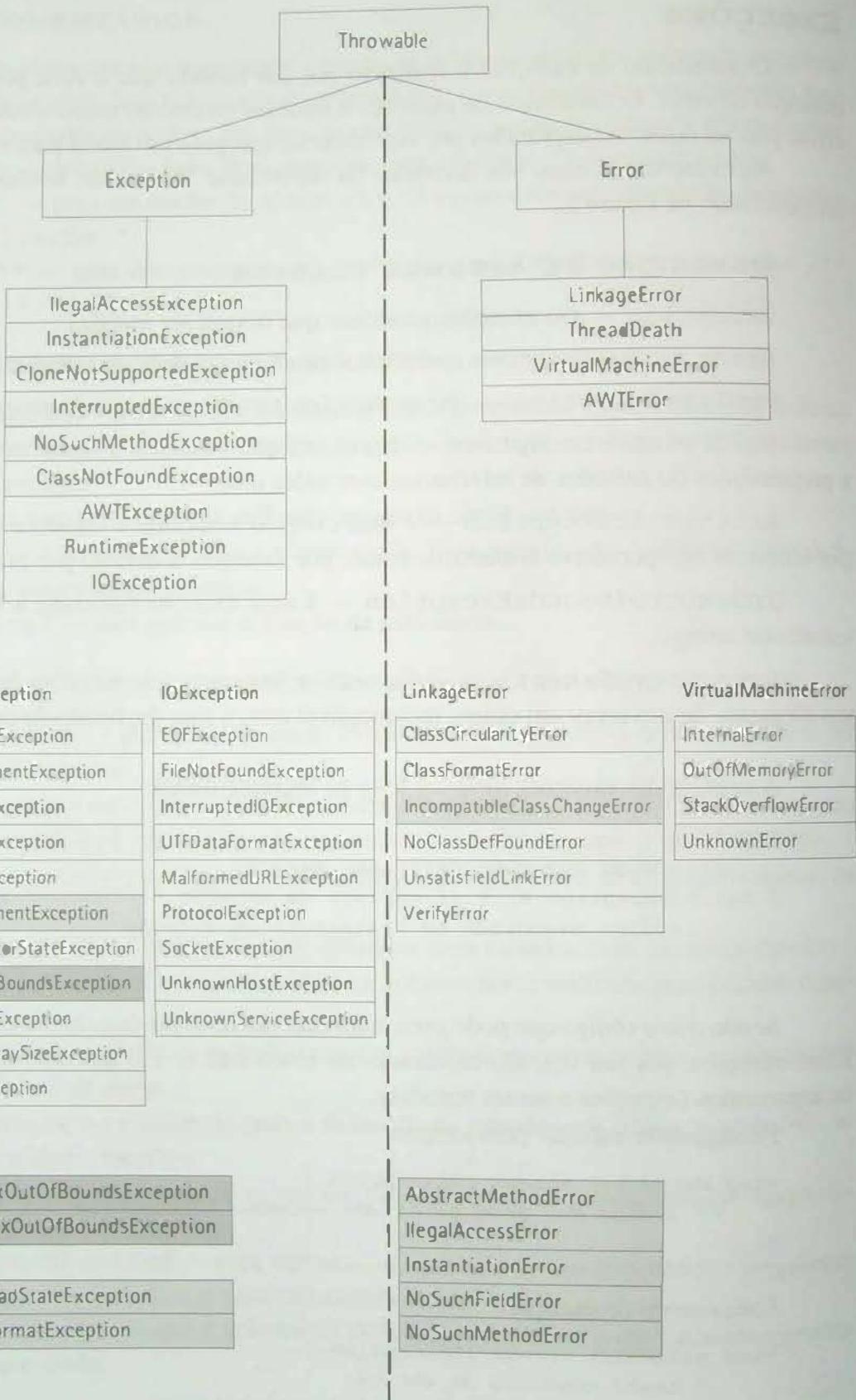


FIGURA 3 | Hierarquia da superclasse Throwable

CLASSE MAIS UTILIZADAS

1. STRING

A classe `String` provê objetos que são mais seguros e simples de utilizar do que os arrays de caracteres. Os objetos `String` são constantes e não podem ser alterados.

Os objetos da classe `String` podem ser inicializados por atribuição direta, isto é, quando se declara um objeto, já se atribui um valor a ele. Por exemplo:

```
String nome = "Rafael";
```

A concatenação de strings é feita utilizando o operador `+`. Por exemplo:

```
nome + " Rissetti"; // o resultado será "Rafael Rissetti" e ficará  
armazenado na variável nome.
```

CONSTRUTORES

`new String(char[] c)` — constrói uma `String` a partir de um array de caracteres.

`new String(String s)` — constrói uma `String` a partir de uma outra `String`.

`new String(StringBuffer sb)` — constrói uma `String` a partir de um `StringBuffer`.

PRINCIPAIS MÉTODOS

`public char charAt(int indice)` — retorna o caractere na posição dada por `indice`.

`public int compareTo(String var)` — compara uma `String` com o parâmetro `String var` e retorna zero se ambas forem iguais, um número negativo se a `String` for menor e um número positivo se for maior.

`public static String copyValueOf(char[] var)` — retorna uma `String` com o conteúdo do array de caracteres `var`.

`public boolean equals(Object o)` — verifica se `o` é uma string com a mesma seqüência de caracteres.

`public boolean equalsIgnoreCase(String var)` — compara uma `String` com a `String var`. As diferenças entre maiúsculas e minúsculas são ignoradas.

`public int length()` — retorna o comprimento da string.

`public int indexOf(int var)` — retorna o índice da primeira ocorrência do caractere `var`.

`public int indexOf(int var, int inicio)` — retorna o índice da primeira ocorrência do caractere `var` a partir de `inicio`.

`public char[] toCharArray()` — retorna um array com os caracteres da string.

`public String toLowerCase()` — retorna uma cópia da string com todas as letras convertidas para minúsculas.

`public String toUpperCase()` — retorna uma cópia da string com todas as letras convertidas para maiúsculas.

`public String trim()` — elimina os espaços em branco no início e no fim da string.

Exemplos:

```
String nome = "RAFAEL";
if(nome.equals("rafael".toUpperCase())) ...
char[] teste = nome.toCharArray();
```

2. CLASSE QUE REPRESENTA DATAS — JAVA.UTIL. DATE

A classe `java.util.Date` representa datas.

CONSTRutoRES

`new Date()` — cria uma data que representa o momento de sua criação.

`new Date(long tempo)` — cria uma data com o tempo em milissegundos após 1º de janeiro de 1970.

PRINCIPAIS MÉTODOS

`public long getTime()` — retorna os milissegundos da data.

`public void setTime(long temp)` — especifica a data com o tempo em milissegundos.

`public boolean before(Date data)` — retorna true se a data é anterior a data.

`public boolean after(Date data)` — retorna true se a data é posterior a data.

`public int compareTo(Date data)` — retorna 0 se a data é igual a data, um número negativo se é anterior e um positivo se é posterior.

Exemplos:

```
Date data1 = new Date();
Date data2 = new Date();
System.out.println("Resposta: " + data1.after(data2));
```

3. CLASSES DE ENTRADA E SAÍDA

`java.io.BufferedReader` — filtro que age sobre um Reader, adicionando um buffer e otimizando o acesso aos dados.

CONSTRUTORES

`new BufferedReader(Reader rd)` — constrói um BufferedReader a partir de um Reader `rd`, com buffer de tamanho default.

PRINCIPAIS MÉTODOS

`public int read() throws IOException` — lê um caractere da entrada-padrão.

`public int read(char [] c, int inicio, int quant) throws IOException` — lê uma sequência quant de caracteres e a coloca no array `c` a partir da posição dada por `inicio`. Retorna o número de caracteres lidos (-1 se não houver caracteres para ler).

`public String readLine() throws IOException` — lê uma linha de caracteres.

`public void close() throws IOException` — fecha o Reader.

Exemplo:

```
BufferedReader arq;
arq = new BufferedReader (new FileReader ("arquivo"));
String linha;
try {
    linha = arq.readLine();
} catch (IOException e) { }
```

`java.io.BufferedWriter` — filtro que age sobre um Writer, adicionando um buffer e otimizando o acesso aos dados.

CONSTRUTORES

`new BufferedWriter (Writer wr)` — constrói um BufferedWriter a partir de um Writer `wr`, com buffer de tamanho default.

PRINCIPAIS MÉTODOS

`public void write(int c) throws IOException` — escreve um caractere na saída.

`public void writer (char [] c, int inicio, int quant) throws IOException` — escreve quant caracteres no array `c` a partir da posição dada por `inicio`.

`public void newLine() throws IOException` — inicia uma nova linha.

`public void flush() throws IOException` — descarrega a saída, forçando a escrita de caracteres que eventualmente estejam no buffer.

`public void close() throws IOException` — fecha o Writer.

Exemplo:

```
BufferedWriter saída;
saída = new BufferedWriter (new FileWriter ("arquivo"));
String linha = "linha do texto";
try {
    saída.write (linha, 0, s.length ());
    saída.newLine ();
} catch (IOException e) { }
```

`java.io.RandomAccessFile` — é um stream de entrada e saída específico para arquivos, cujo acesso não precisa ser seqüencial. Permite especificar se o modo de acesso será somente para leitura ou para escrita e leitura.

CONSTRutores

`public RandomAccessFile(String arq, String modo) throws IOException` — cria um objeto para acesso a arquivo e o associa ao arquivo especificado em arq; modo especifica o modo de acesso que pode ser: `r` (somente para leitura) ou `rw` (leitura e escrita).

`public RandomAccessFile(File arq, String modo) throws IOException` — cria um objeto para acesso a arquivo e o associa ao arquivo especificado por arq; modo especifica o modo de acesso que pode ser: `r` (somente para leitura) ou `rw` (leitura e escrita).

PRINCIPAIS MÉTODOS

`public int read() throws IOException` — lê um byte do arquivo.

`public int read(byte[] x) throws IOException` — lê um array de bytes do arquivo, se houver informações e retorna o número total de bytes lidos, se não retorna -1.

`public int read(byte[] x, int início, int qtd) throws IOException` — lê uma seqüência de qtd bytes e a coloca no array x a partir da posição dada por início. Retorna o número de bytes lidos ou então -1.

`public int readLine() throws IOException` — lê uma linha de caracteres do arquivo.

`public int skipBytes(long x) throws IOException` — descarta x bytes da entrada e retorna número de bytes que foram descartados.

`public void seek(long x) throws IOException` — posiciona o cursor de leitura/escrita na posição x do arquivo (em bytes).

`public long getFilePointer() throws IOException` — retorna a posição corrente do cursor, em bytes.

`public long skipBytes(long x) throws IOException` — descarta x bytes da entrada.

`public long length() throws IOException` — retorna o tamanho do arquivo em bytes.

`public void setLength(long x) throws IOException` — especifica o tamanho do arquivo em bytes.

`public void write() throws IOException` — escreve um byte no arquivo.

`public void write(byte[] b) throws IOException` — escreve um array de bytes no arquivo.

`public void write(byte[] x, int inicio, int qtd)` — escreve a quantidade de bytes determinados por `qtd` bytes do array `x` a partir da posição determinada por `inicio` no arquivo.

`public void close()` — fecha o arquivo.

4. FILE

A classe `java.io.File` representa um arquivo ou um diretório.

CONSTRutores

`new File(String path)` — cria um objeto `File` que representa um arquivo no path especificado.

`new File(String dir, String nome)` — cria um objeto `File` que representa um arquivo no diretório `dir` com o nome especificado.

`public File(File dir, String nome)` — cria um objeto `File` que representa um arquivo no diretório `dir` com o nome especificado.

PRINCIPAIS MÉTODOS

`public boolean canRead()` — verifica se o arquivo pode ser lido.

`public boolean canWrite()` — verifica se o arquivo pode ser escrito.

`public int compareTo(File x)` — compara o path com o de outro arquivo determinado por `x` e retorna 0 se ambos forem iguais.

`public void delete()` — apaga o arquivo associado a esse objeto.

`public boolean exists()` — verifica a existência do arquivo ou diretório.

`public String getName()` — retorna o nome do arquivo ou diretório.

`public String getPath()` — retorna o path do objeto.

`public boolean isDirectory()` — verifica se o objeto representa um diretório.

`public boolean isFile()` — verifica se o objeto representa um arquivo.

- public long length() — retorna o tamanho do arquivo.
- public String[] list() — retorna a lista de arquivos se o objeto representar um diretório.
- public void mkdir() — cria um diretório.
- public void renameTo(File x) — renomeia um arquivo ou diretório de acordo com o nome determinado por x.

Lógica de programação e estruturas de dados

com aplicações em Java

Este livro foi planejado para ensinar lógica, algoritmos e estruturas de dados com exemplos na linguagem Java.

Todos os assuntos são explicados e exemplificados por meio de soluções comentadas, e os exercícios propostos ao final de cada capítulo ajudam a fixar os conteúdos estudados e a verificar o aprendizado.

Este livro mostra passo a passo o uso de estruturas de seleção, repetição, ordenação, ~~funções~~, pilhas, entre outras, implementadas em Java, mas não limita a essa linguagem nem se aprofunda na programação orientada em objetos, descrevendo todos os elementos também em pseudocódigo.

Completam o livro um anexo que apresenta recursos da linguagem Java e um site especial onde estão disponíveis apresentações em PowerPoint e manual de soluções (exclusivos para os professores), as listagens apresentadas no livro e exercícios adicionais para que os alunos possam testar seu conhecimento e aprofundar seus estudos.

Destinado a todos os interessados em programação de computadores, atende principalmente estudantes de graduação em ciência da computação e processamento de dados.

www.prenhall.com/puga_br
site com recursos adicionais
para professores e alunos



www.makron.com.br
www.pearsonedbrasil.com

