Colégio Técnico Industrial Prof. Mário Alquati
Curso Técnico em Informática/Desenvolvimento de Software
Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Lógica de Programação

Leonardo Vianna do Nascimento

Sumário

Índice de ilustrações

Índice de tabelas

Índice de algoritmos

Índice de Programas

1. Introdução

Você com certeza já deve ter observado que os computadores estão por toda a parte. Se ainda não percebeu isso, pare e observe. Hoje, além dos já tradicionais PC's, presentes em praticamente em todos os lugares, temos computadores presentes dentro de celulares, automóveis, aviões e, com a era das TV's digitais chegando, dentro da própria telinha.

Um computador é um dispositivo capaz de realizar vários tipos de operações. Por exemplo, um computador pode funcionar como uma calculadora, como uma máquina de escrever, como uma planilha de cálculos automatizada, como um arquivo de fichas de clientes em uma empresa, como vídeo game, ou até como piloto em um avião comercial.

O que torna isso possível e é o segredo do sucesso destas máquinas é que o computador é capaz de "aprender" todas estas tarefas através de um manual de instruções chamado *programa*. Um programa é uma seqüência de instruções, como uma receita de bolo, que é lida e executada pelo computador, de forma que ele execute a tarefa desejada.

Neste curso iremos aprender como escrever esses programas. Primeiro, porém, vamos conhecer um pouco mais sobre a origem e o funcionamento dessas máquinas incríveis, os computadores.

1.1. A origem dos computadores

Os cálculos matemáticos sempre foram uma necessidade humana. A estrutura de construções, o censo de uma população, o cálculo do valor a ser pago em impostos, tudo necessitava da manipulação precisa de números e cálculos. Instrumentos de cálculo manuais, como o *ábaco* (Figura 1), remontam cerca de 5000 anos.

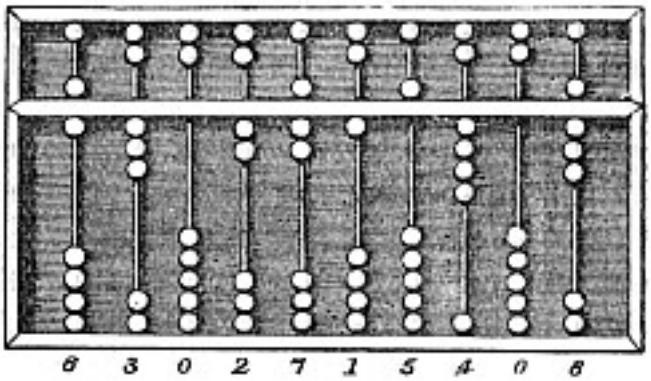


Figura 1: Um ábaco representando o número 6302715408

Apesar dessa necessidade, os primeiros instrumentos automáticos de cálculo só começaram a ser construídos a partir do século XVII. Uma das primeiras calculadoras a serem construídas foi a *Pascalina*, construída pelo matemático *Blaise Pascal* em 1642. Esta calculadora mecânica utilizava um conjunto de engrenagens semelhantes ao contador de quilômetros de um carro para realizar cálculos de soma e subtração.

Entre 1801 e 1805, *Joseph Marie Jacquard* introduziu o conceito de *armazenamento de dados*, utilizando placas perfuradas. Estas placas foram utilizadas no controle de teares mecânicos, utilizados em indústrias de tecelagem. Foram os precursores dos atuais meios de armazenamento, como disquetes, CD's, e outros.

Os projetos mais audaciosos no desenvolvimento de computadores do século XIX foram os desenvolvidos pelo inventor inglês *Charles Babbage*. Ele obteve do governo inglês um financiamento para desenvolver o que chamou de *Máquina Diferencial* (1822 a 1834). Babbage abandonou o projeto quando percebeu que poderia construir algo ainda mais desafiador, a *Máquina Analítica*. A máquina era tão inovadora, que não haviam engrenagens tão precisas na época, o que impossibilitou a sua construção. Entretanto, os projetos de Babbage inspiraram uma série de equipamentos desenvolvidos anos depois, a ponto dele ser conhecido como o "Pai dos computadores".

Até a década de 1930, os computadores construídos eram totalmente mecânicos. Em 1937, foi construído o primeiro computador eletromecânico (parte eletrônico e parte mecânico) chamado *MARK I,* desenvolvido pelo americano Howard Aiken. O primeiro computador totalmente eletrônico e digital foi o *ABC* (*Atanasoff Berry Computer*), criado pelo também americano *John Atanasoff*, a partir de 1937.

Em 1946 foi apresentado o primeiro "grande computador" eletrônico, o *ENIAC – Eletronic Numeric Integrator and Calculator*, que ocupava mais de 170 metros quadrados, pesava cerca de 30 toneladas e funcionava com 18.000 válvulas, sendo capaz de realizar cinco mil somas ou subtrações por segundo (aliás o ENIAC só era capaz de realizar cálculos aritméticos). A preparação do ENIAC para cálculos demorava semanas, pois a programação era realizada pela ligação direta de fios em um painel. Foi desenvolvido na Universidade da Pensilvânia por *John Mauchly* e *J. Presper Eckert*.

O nome seguinte da história dos computadores foi *John von Neummann* que, juntamente com Arthur Burks e Herman Goldstine, desenvolveu entre 1945 e 1950 a lógica dos circuitos, os conceitos de programa e operações com números binários e o conceito de que tanto instruções quanto dados podiam ser armazenados e manipulados internamente (na memória do computador). Suas idéias e conceitos ainda são utilizados muitas décadas depois, nos computadores modernos.

Até a década de 1950, os computadores eram construídos utilizando válvulas eletrônicas. Em 1947, foi desenvolvido o *transistor*, por *William Shockley*, *J. Bardeen* e *W. Brattain*. Seu tamanho era 100 vezes menor do que o da válvula, não precisava de tempo para aquecimento, consumia menos energia, era mais rápido e mais confiável. Os principais computadores construídos com essa tecnologia foram o *IBM 1401* e o *IBM 7094*.

A nova revolução tecnológica viria alguns anos mais tarde, com a invenção dos *circuitos integrados*. O CI (Circuito Integrado) entrou no mercado em 1959, pela *Farchild Semiconductor* e pela *Texas Instruments*, mas só a partir de 1965 começou a substituir o transistor em computadores comercializados. As características dos CI's resumem a evolução e tendências até os dias de hoje: muito mais confiáveis (não têm partes móveis); muito menores (equipamentos mais compactos e mais rápidos pela proximidade dos circuitos); baixíssimo consumo de energia; miniaturização de componentes e muito menor custo. O primeiro computador construído com essa tecnologia foi o *IBM 360*.

A popularização dos computadores começou realmente com a construção dos primeiros *microprocessadores*, a partir da década de 1970. Um microprocessador é um único CI capaz de conter a CPU inteira de um computador (veja a próxima seção para entender o que é uma CPU). Esta nova invenção foi possível devido a utilização de uma nova técnica de miniaturização de

circuitos chamada *VLSI* (*Very Large System Integration*).

O primeiro *microcomputador* (computador utilizando um microprocessador) a ser vendido foi o Altair, que não obteve muito sucesso. A popularização dos microcomputadores ocorreu com o lançamento do *Apple II* pela Apple Computers e pelo *Personal Computer* (PC) pela IBM.

O PC foi um sucesso tão grande que alavancou uma das parceiras da IBM, até então uma empresa de fundo de quintal: a *Microsoft*. A Microsoft forneceu à IBM o *MS-DOS*, o sistema operacional do PC (um conjunto de programas que permite às pessoas utilizarem o computador e seus programas). O MS-DOS foi um produto comercializado até o fim da década de 1990, quando foi substituído pelo *Windows 95,* um sistema operacional gráfico, totalmente desvinculado de seu antecessor.

Atualmente, temos desde pequenos computadores que cabem na palma da mão (*Palmtops*, *Smartphones*) até supercomputadores paralelos, que podem conter milhares de microprocessadores e servem para realizar simulações gigantescas (como o *Earth Simulator*, no Japão, utilizado para simulações climáticas a nível planetário).

1.2. Funcionamento de um Computador

Existem diversas definições para o termo computador. Algumas delas são citadas abaixo:

- "Denomina-se *computador* o conjunto de artifícios eletrônicos capazes de efetuar qualquer espécie de tratamento automático de informações e/ou processamento de dados." (*Wikipedia*)
- "Máquina capaz de receber, armazenar e enviar dados e de efetuar sobre estes, seqüências previamente programadas de operações (como cálculos) e lógicas (como comparações) com o objetivo de resolver problemas." (*Dicionário Aurélio*)

Podemos pensar em um computador como uma máquina capaz de realizar vários tipos de *processamento*. O processamento é uma operação ou transformação sobre dados (entrada) que gera um resultado (saída) visível ao ser humano que o utiliza (veja a Figura 2). Por exemplo, quando uma cozinheira faz um bolo, ela utiliza diversos ingredientes (como farinha, açúcar, ovos, etc). Estes ingredientes são os dados de entrada. O processamento é realizado quando a cozinheira mistura os ingredientes e põe o bolo para assar no forno. O produto final, a saída de dados, apreciado pelos usuários (os felizardos que comerão o delicioso bolo) é o bolo após ter completado seu cozimento no forno (Figura 3).

Figura 2: Processamento de dados efetuado em um computador.

Figura 3: Exemplo de processamento ao se fazer um bolo.

Quando se fala em computadores, ao invés de ingredientes teremos dados de entrada. Como veremos no próximo capítulo, dados são coisas como números (1, -3, 5000), seqüências de caracteres ("CTI", "Rio Grande do Sul"), entre outras coisas. Por exemplo, em um programa de calculadora, os dados de entrada incluem a operação a ser feita (soma, subtração, multiplicação, etc) e os operandos (os números envolvidos na operação). O processamento é o cálculo do resultado da operação e o dado de saída é um número (o resultado da operação, após o calculo/processamento).

A maior parte dos computadores digitais atuais, independentemente de sua aplicação, seguem a arquitetura proposta por John von Neumann, conhecida como *Máquina de von Neumann*. A Figura 4 ilustra a organização dessa máquina.

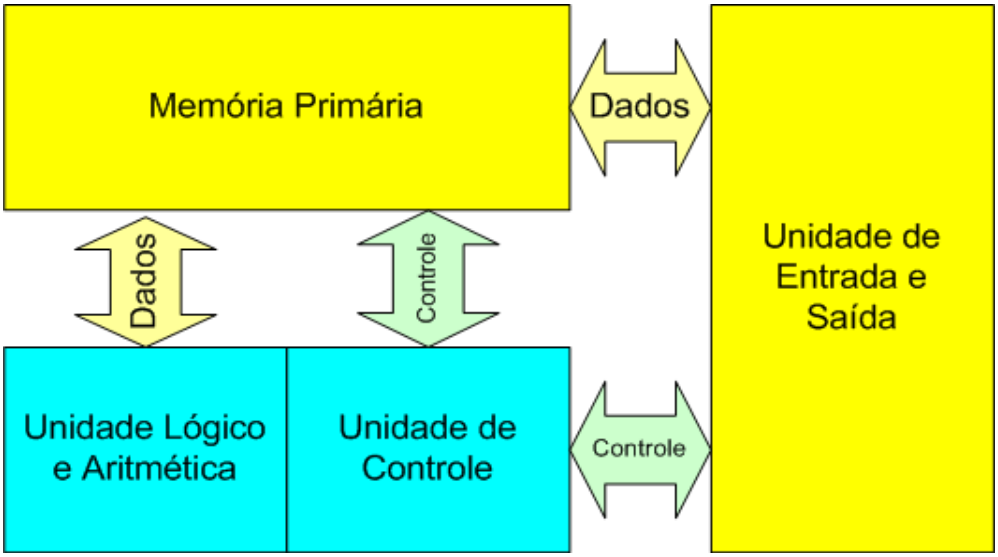


Figura 4: Arquitetura da Máquina de von Neummann

A *Unidade de Entrada e Saída* é responsável por receber as informações necessárias para o processamento (dados de entrada) e também por exibir os resultados deste processamento (dados de saída). Nos computadores modernos esta unidade é composta pelos periféricos de entrada e saída. Podemos citar como exemplos de periféricos de entrada: teclado, mouse, scanner e joystick. Como dispositivos de saída podemos citar a impressora e o monitor de vídeo.

As unidades *Lógico e Aritmética* e de *Controle* formam a chamada *Unidade Central de Processamento* (ou *Central Processing Unit – CPU* – em inglês). Sua função é manipular os dados armazenados na memória e provenientes dos dispositivos de entrada e realizar efetivamente o processamento. Ou seja, a CPU é a unidade responsável pela execução das instruções dos programas. Os programas encontram-se armazenados na memória primária. O componente físico atual que compõe a CPU é o *microprocessador*.

A *Memória Primária* é responsável por armazenar os dados e instruções que serão utilizados no processamento pela CPU. As memórias podem ser classificadas em *voláteis* (memória RAM) e *não voláteis* (memórias ROM, memórias Flash, discos rígidos, CD-RW, etc). A CPU, durante a execução de um programa, manipula freqüentemente a memória, a fim de obter os dados necessários para a execução deste bem como para armazenar dados temporários e os resultados obtidos.

Todos esses componentes são parte do chamado *hardware* do computador. Logo, o hardware é o conjunto de todos os componentes físicos eletrônicos do computador. A Figura 5 mostra alguns componentes do hardware de um microcomputador do tipo PC. A figura apresenta s seguintes ítens:

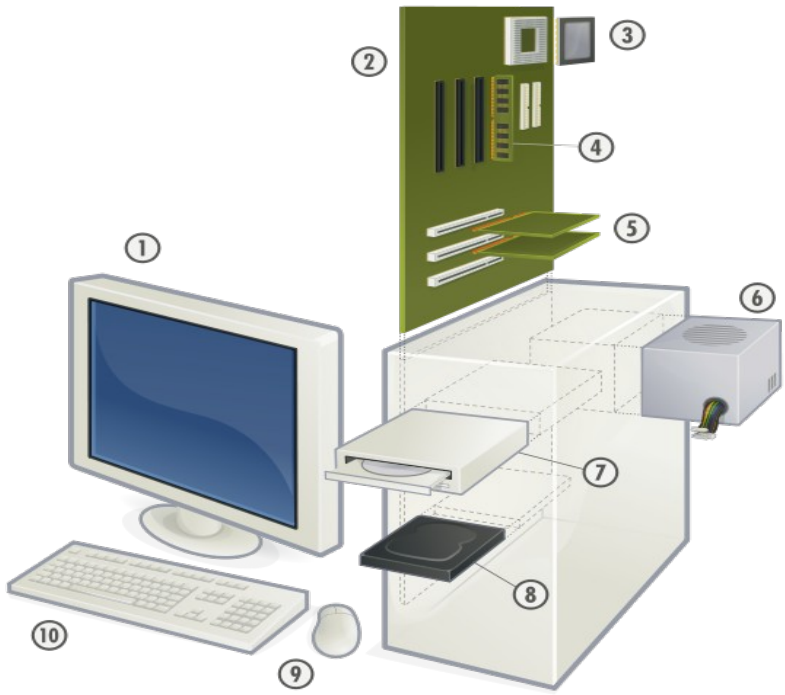


Figura 5: Hardware de um computador pessoal (PC).

- Unidade de Entrada e Saída: (1) *monitor*, (5) *placas de expansão*, (9) *mouse*, (10) *teclado*;
- Memória não volátil: (7) *drive de CD* e (8) *disco rígido*;
- Memória volátil: (4) *memória RAM*;
- CPU: (3) *microprocessador;*
- Outros dispositivos: (2) *placa-mãe* e (6) *fonte de alimentação.*

1.3. Programas de Computador

O *hardware* de um computador precisa ser manipulado de forma coordenada a fim de que o processamento desejado seja feito de forma adequada. Por exemplo, se desejarmos que o computador some dois números automaticamente, precisamos "ensiná-lo"a fazer isso. Isso é feito através do que se chama *software* ou *programa de computador*.

Um programa é uma seqüência de instruções para o computador de como utilizar seus componentes físicos a fim de realizar uma determinada tarefa. Podemos dizer que um programa está para o computador assim como uma receita para uma cozinheira. Sem a receita, a cozinheira não saberá o que fazer com os ingredientes. Assim, também a CPU não sabe o que fazer se não houver um programa que a instrua.

Hoje em dia existem programas que permitem ao computador funcionar como um vídeo-game (jogos de computador), caixa registradora (softwares de ponto de venda em caixas de supermercados e lojas) e até como um atendente bancário (sistemas de auto-atendimento bancário).

Os programas de computador são desenvolvidos por seres humanos. A atividade relacionada ao desenvolvimento de software se chama *programação*. Aquele que programa o software, ou seja, que escreve a receita de bolo para nossa cozinheira CPU, é chamado de *programador*.

Para entender como um programa é executado em um computador, voltemos ao exemplo de nossa cozinheira. Podemos imaginar que os ingredientes são os dados de entrada, a cozinheira é a CPU e a receita do bolo é um programa.

A primeira coisa identificada pela receita são os ingredientes, ou seja, os dados de entrada necessários para que o resultado, o bolo, seja obtido. Sem eles não é possível fazer o bolo. Assim, a maior parte dos programas precisa de dados de entrada, informações externas ao programa que necessitam ser adquiridas pela CPU antes que o processamento propriamente dito seja executado. Por exemplo, um programa que some dois números precisa, antes de executar a soma, obter os dois números a serem somados. Geralmente, como veremos nos próximos capítulos, isso é feito solicitando ao usuário (a pessoa que está utilizando o programa) que informe os números de alguma forma (digitando-os através do teclado, por exemplo).

O próximo passo que a CPU faz é executar as instruções de como fazer o bolo, ou seja, o processamento em si. A receita possui diversas sentenças que podem ser, por exemplo:

```
1. Misture os ingredientes
2. Unte a forma com a manteiga e farinha
3. Despeje a mistura na forma
4. Se houver côco ralado então despeje sobre a mistura
5. Leve a forma ao forno
6. Enquanto não corar deixe a forma no forno
7. Retire do forno
8. Deixe esfriar
```

Veja que as instruções presentes na receita estão exatamente na ordem em que devem ser executadas. Com a receita, qualquer cozinheira pode fazer o bolo especificado. Assim, com a receita, ou seja, o programa, qualquer CPU pode realizar qualquer tarefa. Por exemplo, se quiséssemos que a CPU fosse capaz de somar dois números, poderíamos escrever a seguinte seqüência de instruções:

```
1. Solicite ao usuário que digite um número através do teclado e o
   chame de N1
2. Solicite ao usuário que digite outro número através do teclado e o
   chame de N2
3. Calcule N1 + N2
4. Mostre o resultado da soma no monitor de vídeo
```

Cada instrução desse pequeno programa será executada, uma de cada vez, a partir da primeira. Vamos ver isso na prática? Imaginemos um computador simples, com uma CPU, um teclado, um monitor de vídeo e memória, como mostrado na Figura 6.

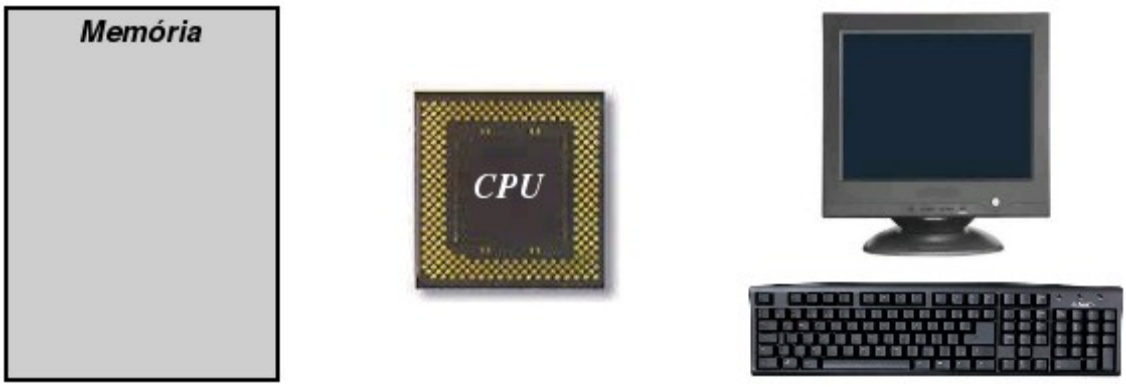


Figura 6: Computador utilizado na execução de nosso pequeno programa.

A primeira instrução a ser executada é a número 1, *Solicite ao usuário que digite um número através do teclado e o chame de N1*. Nessa instrução, a CPU irá esperar que o usuário digite um número através do teclado (*passo 1*). Quando o usuário tiver terminado, a CPU terá conhecimento do número (*passo 2*) e o irá armazenar na memória, para uso posterior (*passo 3*). Como a memória pode armazenar diversos valores ao mesmo tempo, a CPU precisa identificar cada um com um nome diferente. Por isso, o programa instrui a CPU que chame o número digitado de N1. O processo que acabamos de descrever é ilustrado na Figura 7. A segunda instrução é executada de forma similar, mas o número digitado é armazenado na memória com o nome de N2 (Figura 8).

Na terceira instrução, *Calcule N1 + N2*, o programa instrui a CPU de que busque na memória os valores identificados como N1 e N2 e calcule a soma dos dois valores (Figura 9). Ao final, a quarta e última instrução diz a CPU que mostre no monitor de vídeo o resultado da soma (Figura 10).

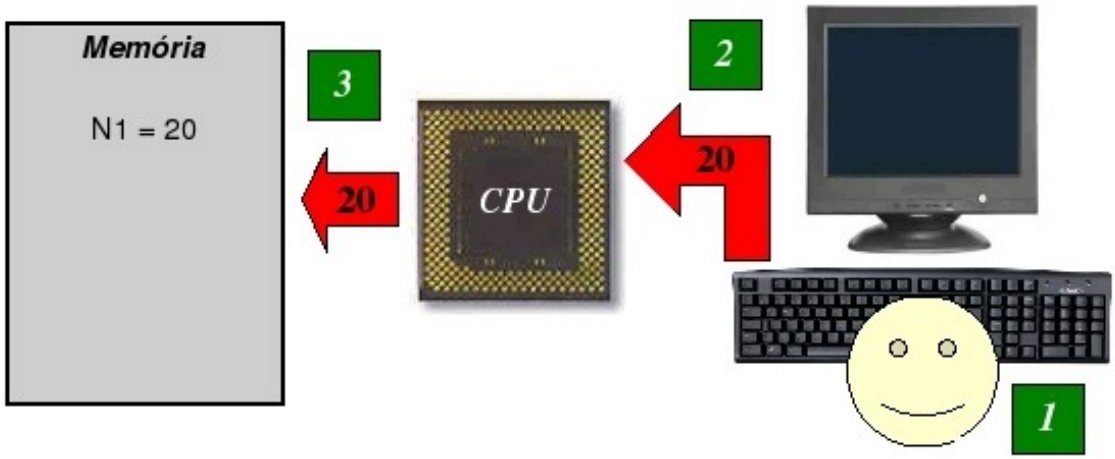


Figura 7: Execução da instrução "Solicite ao usuário que digite um número através do teclado e o chame de N1"

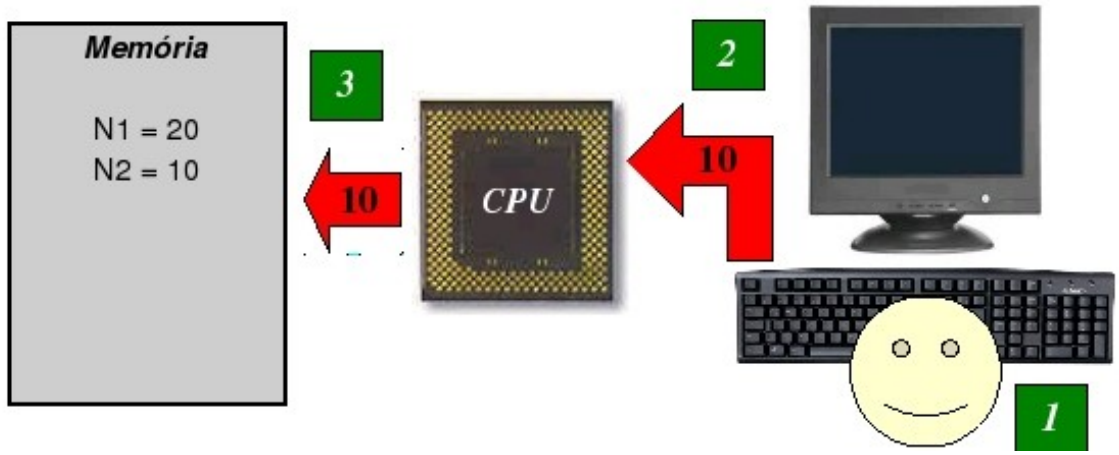


Figura 8: Execução da instrução "Solicite ao usuário que digite outro número através do teclado e o chame de N2"

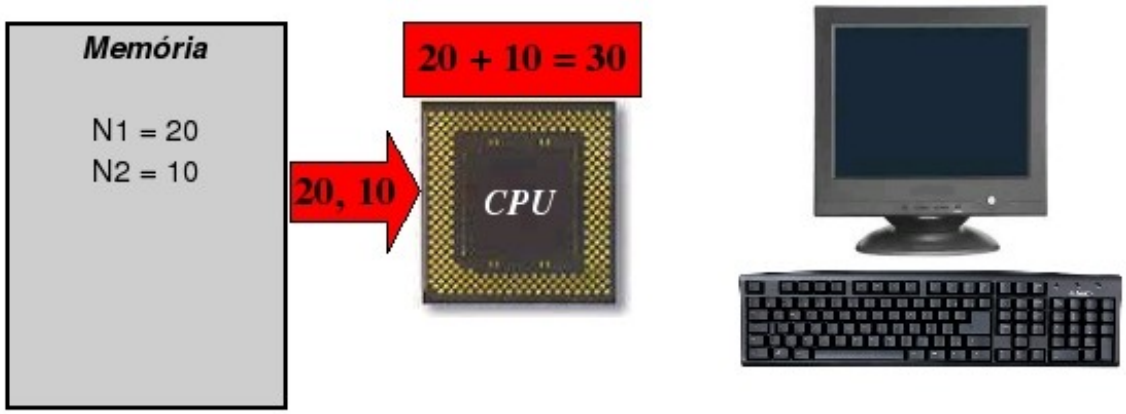


Figura 9: Execução da instrução "Calcule N1 + N2"

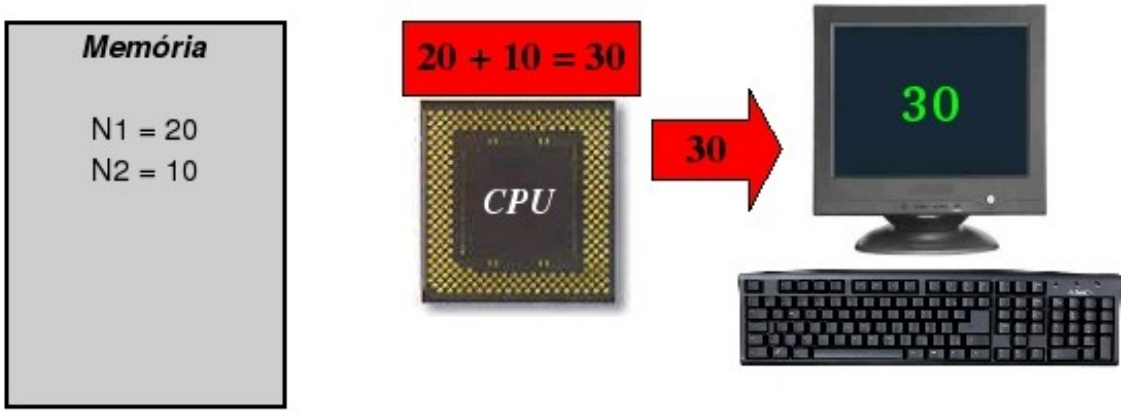


Figura 10: Execução da instrução "Mostre o resultado da soma no monitor de vídeo"

1.4. Criação de um Programa

Apesar de diferirem uns dos outros, os programas têm algo em comum: são todos oriundos de instruções escritas sob determinada lógica, baseada no que o computador deverá fazer. Em outras

palavras: "*o computador só faz aquilo que lhe mandam fazer e não necessariamente o que se deseja que ele faça*". Para que o computador faça o desejado, o programa nele introduzido deve retratar o que realmente se deseja e o que se deve fazer. O programa escrito deverá traduzir fielmente a lógica para solucionar o problema no computador. Desse modo, a expressão: "*o computador errou*" não têm lógica no mundo da informática, pois normalmente quem erra é o *programador*, ou seja, quem escreve programas, e não o computador, afinal o que ele faz é simplesmente executar as ordens contidas nas instruções do programa.

O programa escrito pelo programador é chamado *programa-fonte* ou *código-fonte*. Este programa segue uma seqüência lógica de passos chamada *algoritmo*. Assim, um algoritmo é como se fosse a receita de bolo, um roteiro fiel do que o computador deve fazer para que se obtenha a solução do problema proposto. Então, se o roteiro for mal escrito, o computador executará aquilo que lhe é passado, e a solução não terá a qualidade esperada.

Por isso, a solução de um problema nasce com o algoritmo e não com a execução do programa. O que o computador fará ao executar o programa, é acelerar e automatizar o processo, como uma pessoa lendo uma receita de bolo, e não melhorar a sua qualidade. Então, deve-se sempre lembrar disso: "*a qualidade da solução de um problema proposto a ser resolvido em um computador depende exclusivamente da lógica do algoritmo proposto*".

Todo programa que você utiliza, seja um processador de textos ou um jogo, nasceu a partir de um ou mais algoritmos. Um algoritmo costuma ser escrito de forma que possa ser compreendido por um ser humano, mas não por uma máquina. Como vimos, o hardware de um computador é composto por circuitos eletrônicos, que compreendem sinais elétricos e não a linguagem humana.

Toda CPU existente hoje é capaz de executar uma lista limitada de instruções, como somar dois números, armazenar um número na memória, etc. Cada instrução é representada por um número inteiro positivo diferente. Como veremos no próximo capítulo, isto se deve ao fato de que números podem ser facilmente representados como sinais elétricos dentro da CPU. Logo, a instrução some dois números, em uma CPU particular, pode ser identificada com o número 1, a instrução armazene um número na memória com o número 2 e assim por diante. A este conjunto de instruções básicas da CPU, representadas como números, se dá o nome de *linguagem de máquina*.

Mas como transformar um algoritmo, escrito por exemplo na língua portuguesa, para um programa escrito em linguagem de máquina, que pode ser compreendido pela CPU? No primórdios da programação, esse problema foi resolvido escrevendo-se os algoritmos diretamente em linguagem de máquina. Esta solução se mostrou inadequada com o passar do tempo, a medida que os computadores se tornaram cada vez mais robustos, o que permitiu a escrita de programas mais complicados e maiores. Era necessária uma forma mais simples de transformar algoritmos em programas passíveis de execução pela CPU.

A forma encontrada foi a utilização de *linguagens de programação*. Nestas linguagens, as instruções do programa são representadas por palavras.

As primeiras linguagens de programação utilizadas foram as chamadas *linguagens de baixo nível*, conhecidas como *assembly*. Essas linguagens substituem os números dados às instruções por mnemônicos (conjunto de 3 ou 4 letras que indicam a função da instrução). Assim, por exemplo, a instrução some dois números poderia ser identificada com o mnemônico ADD. Logo um programa que soma-se os números 10 e 20 teria a seguinte instrução em Assembly:

ADD 10, 20

Um programa escrito em assembly precisa ser convertido para a linguagem de máquina da CPU. Este processo se chama *montagem* e é executado por um programa especial chamado *montador* ou em inglês, *assembler*.

Por ser uma linguagem próxima ao código de máquina do computador, o assembly não é apropriada para o desenvolvimento de grandes programas, pois seu código-fonte é de difícil entendimento. Outra desvantagem é o fato de que cada fabricante equipa suas CPUs com uma linguagem de máquina diferente, o que ocasiona um conjunto de instruções assembly diferentes. Um programador que deseja escrever um programa para duas CPUs diferentes, deve escrevê-lo duas vezes, utilizando em cada vez a linguagem de máquina de uma das CPUs.

Por isso, o mais comum hoje em dia é a utilização das chamadas *linguagens de alto nível*. Estas linguagens são mais próximas à linguagem humana. Tais linguagens utilizam algumas palavras (geralmente em inglês) para denominar instruções e estruturas de programação de mais alto nível que são independentes da linguagem de máquina utilizada. Assim, se consegue escrever programas de forma mais clara e simples. Como exemplos de linguagens de alto nível, podemos citar Fortran, Pascal, C, C++ e Java.

Assim, após a elaboração do algoritmo, o desenvolvedor do programa deve reescrevê-lo na linguagem de alto nível de sua escolha. Após esse passo, deve transformar esse programa, escrito em uma linguagem incompreensível pelo CPU, em um programa em linguagem de máquina. Para isso existem dois processos: *compilação* e *interpretação*, ambos conhecidos genericamente como *tradução*.

Na compilação, o programa escrito em linguagem de alto nível é transformado de uma só vez em linguagem de máquina. Este processo é executado por um programa especial chamado de *compilador*. Uma vez compilado, o programa pode ser executado em uma CPU específica quantas vezes for necessário, sem necessidade do código-fonte.

Já na interpretação, o programa é convertido instrução por instrução. O responsável por isso é um programa chamado *interpretador*. O interpretador lê uma instrução do código-fonte, a transforma em código de máquina, e a executa. A cada vez que um programa é executado por um interpretador, é necessário que o código-fonte esteja presente.

As vantagens da tradução por um compilador, é que a execução do programa é mais rápida, pois não é necessário a transformação das instruções em linguagem de máquina a cada execução do programa. A maior parte dos programas presentes em seu computador é compilada. Por outro lado, os compiladores costumam gerar códigos de máquina específicos para determinadas CPUs. Logo, caso se deseje gerar versões do programa para CPUs diferentes, é necessário compilá-lo várias vezes, uma para cada CPU. A vantagem da interpretação, neste caso, é que, presente o interpretador, não é necessário gerar uma versão do programa para a CPU desejada. Basta executar o código-fonte diretamente, através do interpretador. Hoje em dia, os interpretadores são bastante utilizados na execução de programas na Internet.

1.5. Ciclo de Vida de um Software

A criação de um programa, envolve, a rigor, diversas etapas até que seja possível "traduzir" as necessidades do usuário para um sistema computável eletronicamente. É preciso refinar essa realidade para que o computador possa "entendê-la" sem ambigüidades. Em termos gerais, a solução computacional de um problema pode seguir os seguintes passos:

1. **Análise**: Nesta fase o objetivo é obter as necessidades do usuário e construir uma especificação funcional do sistema, ou seja, criar modelos que ilustrem o que o sistema deverá fazer Nesta fase costuma-se definir um *Modelo Conceitual* da solução. Nesse modelo registram-se os dados funcionais do sistema (quais informações estão disponíveis, quais serão as saídas, etc). Este modelo também deve conter os chamados *Requisitos Funcionais*, que especificam tudo o que o sistema deve fazer, servindo para documentar e identificar de forma mais clara o que é realmente necessário.
2. **Projeto**: Após identificar os requisitos do sistema, inicia-se a fase de projeto. Nessa fase partimos do que "deve ser feito" para definir "como fazer". Assim, os modelos construídos aqui especificam qual a melhor forma de realizar as necessidades do usuário no software. Aqui devemos definir os algoritmos e os diagramas lógicos para a solução e definir que tipo de hardware será necessário para a execução do sistema.
3. **Implementação**: É a representação formal dos modelos definidos na fase de projeto em uma linguagem de programação. Uma linguagem de programação é uma linguagem livre de ambigüidades e que pode traduzir de forma mais clara para a linguagem de máquina a lógica especificada nos algoritmos. Os algoritmos escritos em uma linguagem de programação são os códigos-fonte, a partir de onde os programas serão traduzidos para a linguagem de máquina do computador. A fase de implementação será o foco de nosso curso, onde aprenderemos a desenvolver algoritmos e escrevê-los em uma linguagem de programação (o termo técnico para isso é *implementar*).
4. **Teste**: Após implementado, o sistema deve ser testado para identificar possíveis erros durante a sua implementação (conhecidos como *bugs*) e se o produto final atende a todas as necessidades do cliente. Em caso negativo, o sistema deve ser modificado a fim de sanar os problemas identificados, sendo necessário retomar a alguma das fases anteriores e repetir o processo. Caso o produto seja aprovado nos testes, ele é finalmente entregue ao cliente.
5. **Manutenção**: Esta fase é a mais longa do ciclo de um software. Durante a sua utilização, alguns problemas podem ser identificados e o software precisa ser alterado a fim de resolvê-los. Modificações também podem ser solicitadas pelo cliente, a fim de acrescentar um novo requisito que não havia sido implementado durante o desenvolvimento do software.

1.6. Exercícios

1. Complete as lacunas nos textos abaixo:

As _____ foram o primeiro tipo de tecnologia eletrônica a serem utilizadas na construção de CPUs. Hoje em dia não são mais utilizadas, pois durante as décadas de 50 e 60 foram substituídas pelos _____. Hoje em dia, as CPUs são chamadas de _____.

As memórias servem para _____ informações. Já a CPU é responsável pela _____ das instruções de um _____.

As duas formas de se traduzir um programa em linguagem de alto nível são: _____ e _____. Os respectivos programas especiais responsáveis por executá-las são, respectivamente _____ e _____.

Fortran é um exemplo de _____. A linguagem _____ é o único exemplo conhecido de linguagem de baixo nível.

2. Relacione as colunas, marcando na coluna da direita os respectivos termos da coluna da esquerda. Desta forma, relacione cada termo à sua definição, de acordo com o estudado neste capítulo (alguns termos da coluna da esquerda podem não ter correspondente na coluna da direita, e vice-versa).

(1) Hardware	( ) Computador eletrônico construído com cerca de 18.000 válvulas.
(2) Programador	( ) Programa especial que transforma de uma só vez o código-fonte de um programa em código de máquina.
(3) Unidade Lógica e Aritmética	( ) Faz parte da CPU do computador.
(4) ENIAC	( ) CPU encontrada em microcomputadores.
(5) Análise	( ) Dispositivo de saída de dados muito comum em computadores.
(6) Microprocessador	( ) Computador idealizado por Charles Babbage.
(7) Assembly	( ) Conjunto de dispositivos físicos de um computador, como os circuitos eletrônicos.
(8) Código-Fonte	( ) Dispositivo eletrônico que substituiu as válvulas na construção de computadores.
(9) Memória	( ) Linguagem de alto nível.
(10) Impressora	( ) Fase do desenvolvimento de um programa em que se se verificam possíveis erros (bugs) existentes.

3. Classifique os seguintes componentes de hardware como: dispositivos de entrada, dispositivos de saída, dispositivos de armazenamento, CPUs (pesquise, se necessário).

Memória RAM 128 MB; Intel Pentium 4 3.0 GHz; Impressora;

Monitor LCD 17"; Drive de CD-ROM; Pendrive;

Mouse; Caixas de som; Teclado ABNT2 Ergonômico

4. João trabalha em uma empresa de desenvolvimento de software. Ele escreveu um algoritmo e deseja transformá-lo em um programa que rode em seu PC. Para isso ele dispõe de um compilador C++. Quais os passos que ele deve efetuar para transformar seu algoritmo em um programa executável?

5. Muitos fabricantes vendem seus softwares sob licenças que protegem os segredos por trás da solução dos problemas, de forma que um concorrente não possa ter acesso a elas. Por causa disso, os softwares constumam ser vendidos já compilados, e o cliente não tem acesso ao código-fonte. Por que a versão compilada oferece um nível de proteção para o fabricante?

6. José trabalha em uma empresa de consultoria. Em um determinado dia, um cliente apresenta a ele um problema: ele deseja construir um software que esteja disponível para download no servidor central da empresa, de forma que todos os funcionários possam ter acesso a ele e executá-lo em seus computadores. O problema é que não há padronização na configuração dos computadores da empresa: alguns possuem PCs com Windows, outros com Linux, o que impossibilita a utilização de uma única versão compilada para cada computador. Existem duas opções: desenvolver o software utilizando a linguagem C (compilada) ou a linguagem Python (interpretada). Como solucionar o problema em ambos os casos (utilizando C ou Python)?

7. Escreva algoritmos em linguagem natural para solucionar os seguintes problemas:

(a) Calcular um número ao quadrado

(b) Calcular a área de um retângulo, dadas a altura e a base

Mostre a execução de ambos os programas no computador ilustrado na Figura 6 (página 7).

2. Armazenamento de Dados na Memória

No capítulo anterior, aprendemos que a memória é o componente do computador responsável pelo armazenamento dos dados. Aprendemos também que a CPU é quem controla o funcionamento de todos os componentes do computador, através da execução de programas. podemos dizer que a memória é o bloco de rascunho da CPU, onde ela toma nota do que deseja lembrar mais tarde.

Vimos também que como todo o hardware, a memória é um componente eletrônico. É fácil imaginar alguém tomando nota em um bloco de papel, mas como tomar nota em um componente eletrônico, em meio a um emaranhado de circuitos eletrônicos minúsculos? Pois neste capítulo esta pergunta será respondida e veremos como é possível utilizar a memória para armazenar dados.

2.1. Informação e Dado

"*Informação é o resultado do processamento, manipulação e organização de dados de tal forma que represente um acréscimo ao conhecimento da pessoa que a recebe*" (Wikipedia).

Podemos dizer que dado é tudo aquilo que pode ser processado, ou seja, números, medições, valores lógicos (verdadeiro e falso), textos, etc. Informações são os dados agrupados de forma a agregar conhecimento, ou seja, são os dados agrupados de tal forma a significar algo útil às pessoas que o lêem.

Por exemplo, o número "30" é um dado, pois pode ser processado (em um cálculo, por exemplo) mas não significa nada isoladamente. Mas se dissermos "*Hoje a temperatura máxima atingiu 30 ºC*" organizamos os dados de forma a obter informação, pois agora existe um significado.

Os programas de computador são capazes de manipular dados. Apesar disso, a interface deles com o usuário humano deve ser capaz de gerar informações a partir desses dados, a fim de que os dados gerados pelo processamento possam ser entendidos pelos seres humanos.

2.2. Tipos Primitivos de Dados

Os programas de computador são capazes de manipular dados de vários tipos. Nesta seção introduziremos alguns tipos existentes em diversas linguagens de programação e que utilizaremos nos algoritmos em pseudocódigo. Estes tipos são: *inteiro*, *real*, *lógico*, *caractere* e *literal*. Estes tipos são chamados primitivos pois já estão disponíveis para uso. Veremos mais tarde que o próprio programador pode criar tipos de dados diferentes dos primitivos.

2.2.1. Tipo Inteiro

Valores deste tipo representam todos os números positivos, negativos e o zero, sem casas decimais, o que exclui os números fracionários e irracionais. Por exemplo, 24, 0 e -12 são exemplos de valores do tipo inteiro.

Estes números, apesar de serem representados matematicamente na classe dos números reais, são classificados como dados do tipo inteiro, por não possuírem parte fracionária. Esta possibilidade é interessante por permitir uma economia do espaço de memória, como veremos adiante.

2.2.2. Tipo Real

Representa os dados numéricos que possuem parte fracionária, positivos e negativos. Baseado nas linguagens de programação, utilizaremos como separador decimal o ponto (“.”) e não a vírgula. Isto porque as linguagens de programação utilizam o padrão norte-americano para notação de números reais. Como exemplos de dados reais temos: 3.14159265, -9.806, 0. (zero sem casas decimais), 12.0.

Observe que há uma diferença entre '0', que é um dado do tipo inteiro, e '0.' (ou '0.0') que é um dado do tipo real. Portanto, a simples existência do ponto decimal serve para diferenciar um dado numérico do tipo inteiro de um do tipo real.

2.2.3. Tipo Lógico

O tipo lógico (também chamado Booleano) define apenas dois valores: .V. (que representa verdadeiro) e .F. (que representa falso). Os pontos fazem parte da notação dos valores e não devem ser omitidos. Assim, “V” e “F”, sem os pontos, não serão considerados valores lógicos nos algoritmos.

2.2.4. Tipo Caractere

Representa símbolos alfanuméricos, como letras ou algarismos numéricos. Sinais de pontuação e qualquer outro símbolo válido também se enquadram neste tipo. Os valores do tipo caractere devem ser representados entre aspas simples (' '). Por exemplo, os valores 'A', '9', '@' e '?' são dados do tipo caractere.

2.2.5. Tipo Literal

Este tipo de dado, também chamado de *cadeia de caracteres* ou *string*, armazena uma seqüência de caracteres alfanuméricos. Sua representação deve ser entre aspas duplas (“ “).

O *comprimento* de um dado literal é o número de caracteres que o compõe. Como exemplo de dados literais temos: “CTI” (comprimento 3), “Silva” (comprimento 5), “ABC123” (comprimento 6), “1.2” (comprimento 3).

Note que, por exemplo, “1.2” representa um dado do tipo literal de comprimento 3, constituído pelos caracteres '1', '.' e '2', diferindo de 1.2 que é um dado do tipo real. Qualquer coisa que estiver entre aspas duplas, mesmo que seja um número, será considerado um dado do tipo literal.

2.3. Representação de Dados na Memória do Computador

A todo momento durante a execução de qualquer tipo de programa, os computadores estão manipulando informações representadas pelos diferentes tipos de dados. Para que não se “esqueça” das informações, o computador precisa guardá-las em sua memória, o seu bloco de rascunho particular.

2.3.1. Organização da Memória

A memória de um computador pode ser vista, de forma bastante simplificada, como um conjunto ordenado de células. Podemos visualizar as células de memória como linhas em um caderno. Cada uma destas linhas recebe um número inteiro positivo diferente, conhecido como endereço (Figura 11). Atualmente, as memórias podem conter até bilhões destas células!

Endereço	Dado
0	
1	
2	
3	
4	
5	
...	...

Figura 11: Representação simplificada das células de memória de um computador.

Quando a CPU deseja armazenar algum dado, seja de qual tipo for, ela o escreverá na memória utilizando um ou mais endereços, assim como nós, quando desejamos escrever algo em um caderno, utilizamos uma ou mais linhas. O número de células (linhas) utilizadas vai depender do tamanho do que desejamos escrever e do que cabe em cada célula. Veremos adiante que alguns tipos de dados, como lógico por exemplo, necessitam de apenas uma célula para escrita, enquanto dados do tipo literal costumam utilizar mais de uma célula, dependendo de seu comprimento.

Cada célula de memória é mais conhecida como *byte*. Como mostrado na Figura 12, um byte é um agrupamento de 8 *bits*, numerados em ordem crescente, da direita para a esquerda, de 0 até 7. Diz-se que o bit 7 é o mais significativo e o bit 0 é o menos significativo.

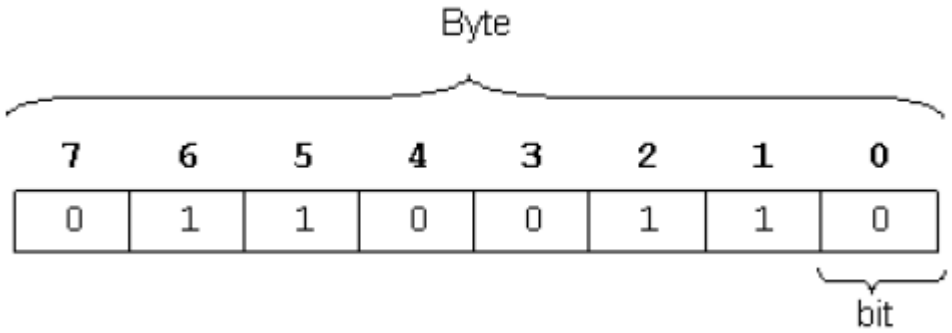


Figura 12: Representação de um byte na memória de um computador.

Por sua vez, cada bit é capaz de assumir dois valores possíveis: 0 e 1, ou desligado e ligado, respectivamente. Um bit é a base da informação computacional. Independente de suas representações físicas, ele sempre é lido como 0 ou 1. Uma analogia a isso são as posições de um interruptor de luz – a posição desligada pode ser representada por 0, enquanto sua posição ligada pode ser representado por um.

Por que apenas esses dois valores? Fisicamente, o valor de um bit é, de uma maneira geral, armazenado como uma carga elétrica acima ou abaixo de um nível padrão em um único capacitor dentro de um dispositivo de memória.Mas, bits podem ser representados fisicamente por vários meios. Os meios e técnicas comumente usados são: pela eletricidade, como já citado, por via da luz (em fibras ópticas, ou em leitores e gravadores de discos ópticos por exemplo), por via de ondas eletromagnéticas (rede wireless), ou também, por via de polarização magnética (discos rígidos). A

manipulação de dois estados (níveis) possíveis para um bit se mostrou mais eficaz, e é utilizada até hoje.

Logo, podemos imaginar um byte como uma seqüência de oito interruptores de luz, que podem estar ligados ou desligados, ou como um conjunto de oito lâmpadas, cada uma acesa ou apagada. A CPU altera o valor contido em um byte ligando ou desligando um ou mais de seus bits. É mais ou menos como em um semáforo, em que a ligando a luz vermelha significa PARE, ligando a luz amarela significa ATENÇÃO e ligando a luz verde significa SIGA. Veremos na próxima seção como é possível representar valores através de um conjunto de bits.

O byte é utilizado também como unidade de medida do tamanho de agrupamentos de dados. Neste caso, como acontece com outras unidades, como metro, grama, e outras, temos algumas unidades derivadas conforme a Tabela 1.

Unidade	***Símbolo***	***Equivalente a***
Kilobyte	KB	1024 bytes
Megabyte	MB	1024 kilobytes ou 1.048.576 bytes
Gigabyte	GB	1024 megabytes ou 1.073.741.824 bytes

Tabela 1: Unidades derivadas do byte.

Apesar dos bytes serem subdivididos em pedaços menores, os bits, a menor porção de memória acessível é o byte. Em outras palavras, se quisermos escrever (ou ler) algum dado na (da) memória do computador, teremos de fazê-lo, no mínimo, byte a byte.

Nos computadores modernos, que trabalham com grandes volumes de dados (muitas vezes na ordem de megabytes ou até gigabytes), permite-se trabalhar com pequenos agrupamentos de bytes chamados *palavras*. O tamanho de cada palavra depende do tipo de hardware utilizado na máquina. Atualmente é comum encontrarmos computadores com palavras de 32 bits (4 bytes) e mais recentemente de 64 bits (8 bytes).

2.3.2. Sistemas Binário

Como vimos no capítulo anterior, as CPUs lidam com instruções na forma de números. Isto não se restringe só às instruções, mas os dados manipulados pela CPU também são números.

Logo, os valores armazenados na memória também devem ser números, pois são diretamente manipulados pela CPU. Como vimos na seção anterior, a memória é formada por um conjunto de bytes que, por sua vez, são formados por um conjunto de 8 bits. Um bit pode ter apenas dois valores possíveis: ligado ou desligado, 0 (zero) ou 1. Então, como representar um número utilizando bits?

Todos os dados armazenados nos bytes da memória são números representados eletronicamente. Esta forma de representação difere da que utilizamos no dia-a-dia.

Segundo a Wikipedia, um *numeral* é um símbolo ou grupo de símbolos que representa um número. Os numerais diferem dos números do mesmo modo que as palavras diferem das coisas a que se referem. Os símbolos "11", "onze" e "XI" são numerais diferentes, representando todos o mesmo número. Ou seja, os numerais são formas diferentes de se dizer a mesma coisa.

Um *sistema de numeração*, (ou *sistema numeral*) é um sistema em que um conjunto de números são representados por numerais de uma forma consistente. Por exemplo, no sistema de numeração romano, os números *três* e *onze* são representados respectivamente por III e XI, enquanto no sistema decimal por 3 e 11. Um sistema de numeração define regras de como os números serão representados através de símbolos. Estes símbolos são chamados *algarismos*.

Atualmente, utilizamos o *sistema decimal*. Este sistema utiliza dez algarismos (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) para se construir os numerais que representam qualquer número. Conseguimos construir qualquer número utilizando apenas 10 dígitos porque trabalhamos com uma *notação posicional*. Isso significa que a posição de um dígito em um número especifica o valor desse dígito no número. Desse modo, quando uma pessoa diz que possui 52 CD's, o zero significa que não existe nenhuma unidade e o 5 significa que existem 5 dezenas. Numerando os dígitos, a partir de 0, da direita para esquerda temos que:

$$5_1 2_0 = 5 \times 10^1 + 2 \times 10^0$$

Para se saber o valor de um dígito em um número, no sistema decimal, basta multiplicar o algarismo daquele dígito por dez elevado ao número de sua posição. Por exemplo, o dígito 5, no número 52 mostrado acima, está na posição 1. Para sabermos seu valor na representação do número multiplicamos 5 por 10 elevado a 1, obtendo 50. Para sabermos o valor do número, a partir de um numeral decimal, basta somar os valores de todos os dígitos, após a repetição do processo descrito para cada um (como mostrado acima). Os valores de outros números, como 132 e 5078, podem ser obtidos pelo mesmo processo:

$$1_2 3_1 2_0 = 1 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$$

$$5_3 0_2 7_1 8_0 = 5 \times 10^3 + 0 \times 10^2 + 7 \times 10^1 + 8 \times 10^0$$

A *base* de um sistema de numeração é o número de algarismos utilizados na representação do número. O sistema decimal, por utilizar dez algarismos, possui base 10. Esse é o motivo de utilizarmos o número 10 nas multiplicações acima, para obter os valores de cada dígito nos números.

Existem sistemas de numeração em que são utilizados menos algarismos ou mais algarismos. Por exemplo, existem os sistemas *octal* (que utiliza 8 algarismos) e o sistema *hexadecimal* (que utiliza 16 algarismos). Logo, podemos ter sistemas de numeração que utilizem qualquer número de algarismos.

Como representar os números através de bits que podem apenas assumir dois valores? Simples, basta utilizar um sistema de numeração que utilize apenas dois algarismos. Este sistema existe e é chamado de *sistema binário*. Este sistema de numeração representa os números utilizando apenas os algarismos 0 e 1, possuindo base 2. Assim, cada número armazenado em um byte é representado no sistema binário, onde cada bit representa um dígito do número. Geralmente o algarismo 0 representa o estado desligado do bit e o algarismo 1 o estado ligado.

O sistema binário utiliza a notação posicional, como a que vimos no sistema decimal. Assim temos, por exemplo:

$$1_1 1_0 \text{ (base 2)} = 1 \times 2^1 + 1 \times 2^0 = 3$$

$$1_3 1_2 0_1 1_0 \text{ (base 2)} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$$

Como o sistema binário utiliza a base 2, as multiplicações na obtenção dos valores dos dígitos são feitas utilizando-se o número 2, e não o número 10. A Tabela 2 mostra a correspondência entre alguns números escritos na base 10 e base 2.

Decimal (base 10)	***Binária (base 2)***
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Tabela 2: Alguns números representados nas duas bases.

Como vimos, é fácil transformar um numeral binário em um numeral decimal. Mas como transformar um numeral decimal em um numeral binário?

Para transformarmos um numeral binário para um decimal utilizamos o *método do caminho inverso dos restos das divisões sucessivas*. Nesse método, fazemos as divisões sucessivas do número na base 10 por 2, pois os numerais binários estão na base 2. Guardamos o resto obtido e dividimos o resultado dessa divisão novamente por 2. Repetimos essa operação até que o resultado da divisão seja zero. Nesse momento, então, escrevemos todos os restos das divisões do fim para o início, de forma que o primeiro resto será o último dígito do número na base 2 e o último resto será o primeiro dígito. Por exemplo, para transformarmos o número 11 para a base 2 temos:

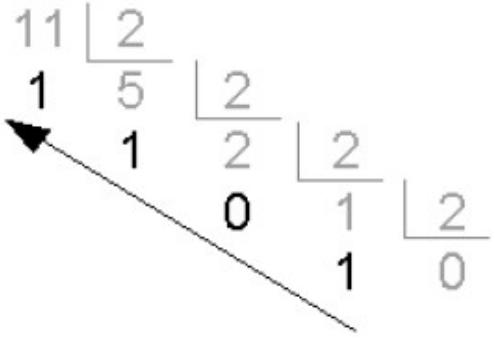


Figura 13: Transformação do número onze representado no sistema decimal para o sistema binário.

Logo, o número 11 é representado no sistema binário pelo numeral 1011. De forma semelhante, podemos fazer a conversão de outros números.

2.3.3. Armazenamento de Dados na Memória

Todos os dados armazenados nas células de memória do computador são números no formato binário. Cada dígito do número é representado dentro de um bit. Sendo assim, cada byte (que contém 8 bits) pode armazenar, no máximo, números com 8 dígitos binários.

Dados do tipo inteiro e real são facilmente armazenados na memória, já que representam números. Números que são representados por mais de 8 dígitos no sistema binário são armazenados em vários bytes consecutivos. A Figura 14 mostra como os números 9 (1001 em binário) e 1234 (10011010010 em binário) podem ser armazenados na memória. Cada dígito binário em cada linha representa um bit. Há oito dígitos em cada linha, pois cada byte contém 8 bits. Como no sistema decimal, zeros à esquerda de um numeral binário não alteram seu valor.

Endereço	Dado
...	
10	0 0 0 0 1 0 0 1 (número 9)
11	
12	1 1 0 1 0 0 1 0 (número 1234, dígitos 0 a 7)
13	0 0 0 0 0 1 0 0 (número 1234, dígitos 8 a 15)
14	
...	

Figura 14: Armazenamento dos números inteiros 9 e 132 na memória.

Mas como são armazenados os demais tipos? Basta representá-los como números.

Dados do tipo lógico representam apenas dois valores. Assim, pode-se associar, por exemplo, o valor 0 para falso e 1 para verdadeiro. Essa associação na prática dependerá da linguagem de programação utilizada.

Os dados do tipo caractere utilizam uma codificação numérica. Cada caractere é associado a um número (código) diferente. Hoje em dia existem diversas codificações para caracteres.

Uma das codificações mais conhecidas para representação de caracteres em computadores, na forma de números, é o chamado padrão ASCII. Alguns caracteres e seus respectivos códigos são mostrados na Tabela 3. Nesta codificação, cada caractere é representado por um número distinto, entre 0 e 255. Assim, para se armazenar um caractere utilizando a tabela ASCII precisamos de apenas um byte.

Caractere	*Valor Decimal*	*Caractere*	*Valor Decimal*	*Caractere*	*Valor Decimal*
Branco	32	, (vírgula)	44	8	56
!	33	-	45	9	57
“	34	. (ponto)	46	A	65
#	35	/	47	B	66
$	36	0	48	C	67
%	37	1	49	D	68
&	38	2	50	E	69
'	39	3	51	a	97
(	40	4	52	b	98
)	41	5	53	c	99
*	42	6	54	d	100
+	43	7	55	e	101

Tabela 3: Alguns caracteres e sua codificação utilizando a tabela ASCII.

Um dado do tipo literal nada mais é do que uma seqüência de caracteres. Assim, sua representação na memória consiste em uma seqüência de códigos numéricos de caracteres, como mostrado na Figura 15. A figura mostra o caso em que se armazena o literal “banana” no conjunto de seis bytes contíguos de memória, iniciando pela posição de memória 100. Na verdade, ao invés dos caracteres do literal, os códigos correspondentes aos mesmos é que são guardados na memória.

Endereço	**Dado**
...	
100	b(98) - 0 1 1 0 0 0 1 0
101	a(97) - 0 1 1 0 0 0 0 1
102	n(110) - 0 1 1 0 1 1 1 0
103	a(97) - 0 1 1 0 0 0 0 1
104	n(110) - 0 1 1 0 1 1 1 0
105	a(97) - 0 1 1 0 0 0 0 1
...	

Figura 15: Armazenamento do literal "banana" na memória do computador.

2.4. Variáveis e Constantes

Como já vimos, a memória de um computador pode ser vista como um conjunto de células onde dados de diversos tipos podem ser armazenados. Cada uma das células de memória é identificada por um endereço numérico. Assim, para acessarmos um determinado dado da memória necessitaríamos conhecer o tipo deste dado (para sabermos o número de bytes de memória ocupados por ele), bem como a posição inicial deste conjunto de bytes na memória.

Percebe-se que esta sistemática de acesso a informações na memória é bastante difícil de se trabalhar. Imagine termos que, a cada execução de um programa, identificar o endereço de memória onde estão os nomes de cada um de 3.000 funcionários de uma empresa! Para contornar esta situação criou-se o conceito de *variável*, que é uma entidade destinada a guardar uma informação.

Cada variável representa uma ou mais posições de memória onde um determinado dado encontra-se armazenado. Quando desejamos acessá-lo precisamos saber somente qual a variável que o contém, e não mais seu endereço de memória. Lembre-se dos programas simples que vimos no capítulo anterior. Para cada dado armazenado na memória foi dado um nome diferente. Este nome representa uma variável.

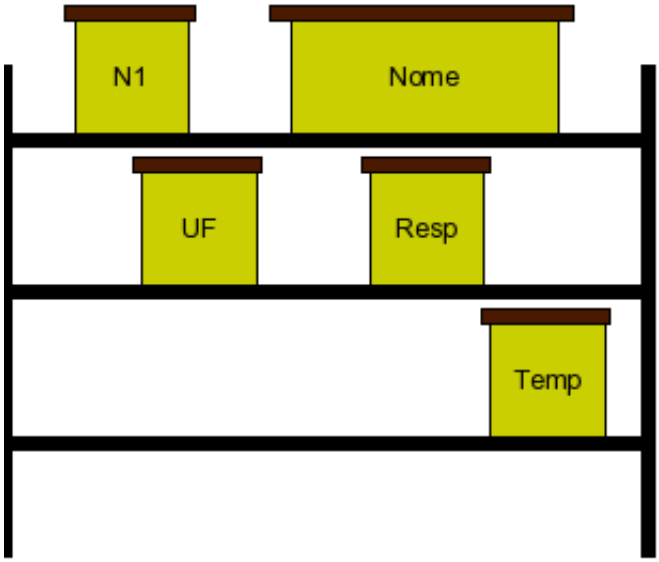


Figura 16: Modelo de memória utilizando variáveis como caixas em uma estante.

No modelo de variáveis, não trabalhamos mais com seqüências de endereços. Neste modelo, a memória pode ser representada como uma estante, com diversas caixas. Cada caixa representa uma variável, e pode armazenar algum dado ou um conjunto deles na memória, como mostrado na Figura 16. Quando a CPU deseja colocar um dado em uma variável, ela abre a caixa com aquele nome e substitui o que há lá dentro pelo novo dado. Quando ela deseja consultar o valor da variável, procura pela caixa correspondente na memória e verifica o que há lá dentro.

As variáveis serão definidas dentro dos programas. Isso quer dizer que cada programa deve especificar à CPU quais variáveis irá utilizar de forma clara. Veremos como fazer isso mais adiante.

2.4.1. Atributos de uma Variável

Basicamente, uma variável possui três atributos: um *nome*, um *tipo de dado* associado à mesma e o *dado* por ela guardado.

Toda variável possui um *nome* que tem a função de diferenciá-la das demais. Cada linguagem de programação estabelece suas próprias regras de formação de nomes de variáveis. Adotaremos nesta disciplina as seguintes regras (baseadas na linguagem Java):

1. um nome de variável deve necessariamente começar com uma letra ou um sublinhado (_) ;
2. um nome de variável não deve conter nenhum símbolo especial, exceto o sublinhado;
3. após o primeiro caractere podem ser empregados dígitos numéricos (0–9);
4. não pode haver espaços em branco entre os caracteres;
5. devem ser evitadas letras com acentuação (ex.: á, à, â, ã) e o cedilha (ç).

Exemplos:

```
SALARIO     - correto
1ANO        - errado (não se deve começar com um dígito numérico)
ANO1        - correto
A CASA      - errado (contém o caractere branco)
SAL/HORA    - errado (contém o caractere '/')
SAL_HORA    - correto
_DESCONTO   - correto
```

Obviamente é interessante adotar nomes de variáveis relacionados às funções que serão exercidas pelas mesmas dentro de um programa. Por exemplo, para guardar o salário de um funcionário de uma empresa poderíamos utilizar uma variável chamada SALARIO, ao invés de chamá-la X ou Y, que não tornam claro o que estamos armazenando.

Um outro exemplo seria o de identificar uma variável que armazenasse o valor da devolução do imposto de renda do ano base de 2005. Neste caso, poderíamos optar por qualquer um dos nomes abaixo:

```
IR
DevIR
DevIR05
Devolucao_do_Imposto_de_Renda_de_2005
```

Qual desses identificadores seria o mais adequado? O primeiro parece muito curto; o segundo parece mais prático que o primeiro, pois identifica uma devolução (com o emprego do prefixo Dev). O terceiro é mais explícito, pois indica que a devolução é do ano 2005. Já o quarto nome é o mais explícito de todos, mas é muito longo! Assim, o nome que mais se aproxima do ideal

é o terceiro: **DevIR05**, pois define de forma clara e suscinta o conteúdo da variável. Portanto, o nome de uma variável vai depender diretamente do bom senso do programador.

Outro atributo característico de uma variável é o *tipo de dado* que ela pode armazenar. Este atributo define a natureza das informações contidas na variável. Dependendo do tipo de dado, a variável pode ser maior ou menor, ou seja, ocupar mais espaço (bytes) em nossa estante (memória). Por último, há o atributo *dado*, que nada mais é do que o dado contido na variável.

Uma vez definidos, os atributos nome e tipo de uma variável não podem ser alterados durante a execução de um programa. Por outro lado, o atributo dado está constantemente sujeito a mudanças, de acordo com a fluxo de execução do programa. Por exemplo, se definirmos que uma determinada variável chamada SALARIO é destinada a guardar números reais, é possível que seu conteúdo seja, num dado instante, igual a 1500.00 e posteriormente modificado para 3152.19, de acordo com o programa executado.

2.4.2. Declaração de Variáveis em Algoritmos

Todas as variáveis utilizadas em algoritmos devem ser declaradas antes de serem utilizadas. Declarar uma variável é dizer, *a priori*, à CPU o nome e o tipo de cada variável utilizada. Isto se faz necessário para que a CPU reserve um espaço na memória para as mesmas.

Dependendo da linguagem de programação empregada, a declaração de variáveis pode mudar. Nesta disciplina será adotada a seguinte convenção nos algoritmos:

- todas as variáveis utilizadas em algoritmos serão declaradas no início do mesmo, por meio de um comando, de uma das formas seguintes:

```
VAR <nome_da_variável> : <tipo_da_variável>;
ou
VAR <lista_de_variáveis> : <tipo_das_variáveis>;
```

- a palavra-chave **var** deverá estar presente sempre e será utilizada uma única vez na definição de um conjunto de uma ou mais variáveis;
- numa mesma linha poderão ser declaradas uma ou mais variáveis do mesmo tipo; para tal deve-se separar os nomes das mesmas por vírgulas;
- variáveis de tipos diferentes devem ser declaradas em linhas diferentes;
- linhas de declaração de variáveis devem ser separadas por ponto-e-vírgula (;).

Exemplo de definição de variáveis:

```
var       Nome, Endereco : literal;
          Idade          : inteiro;
          Salario        : real;
          Tem_Filhos     : lógico;
```

No exemplo acima foram declaradas cinco variáveis:

- as variáveis **Nome** e **Endereco**, capazes de armazenar dados literais;
- a variável **Idade**, capaz de armazenar um número inteiro;
- a variável **Salario**, capaz de armazenar um número real;
- a variável **Tem_Filhos**, capaz de armazenar uma informação lógica.

Um outro exemplo: Declarar variáveis para armazenar a devolução do imposto de renda de 2005, nome de um cliente, total de vendas no mês de dezembro, resposta do usuário se confirma ou não a exclusão de um cliente do arquivo e o total de alunos do sexo feminino de um colégio.

A solução para este exercício poderia ser:

```
var  DevIR05        : real;
     NomeCliente    : literal;
     TotVendasDez   : real;
     Resposta       : lógico;
     TotAlunosFem   : inteiro;
```

2.4.3. Constantes

As constantes, assim como as variáveis, representam simbolicamente endereços de memória. Entretanto, ao contrário de uma variável, o valor armazenado em uma constante não pode mudar durante o processamento de um programa. Quando declaramos uma constante, devemos imediatamente atribuir um valor para ela, o qual permanecerá inalterável durante o fluxo de execução do programa. Quanto aos outros atributos (nome e tipo de dado), estes funcionam de forma idêntica às variáveis.

Para declararmos uma constante utilizamos uma das seguintes formas:

```
CONST <nome_da_constante> : <tipo_da_variável> = <valor>;
```

ou

```
CONST <lista_da_constante> : <tipo_das_variáveis> = <valor>;
```

- a palavra-chave **const** deverá estar presente sempre e será utilizada uma única vez na definição de um conjunto de uma ou mais constantes;
- constantes devem ser declaradas em linhas diferentes;
- linhas de declaração de variáveis devem ser separadas por ponto-e-vírgula (;).

2.5. Exercícios

1. Classifique os dados abaixo marcando **L** para literal, **C** para caractere, **I** para inteiro, **R** para real e **B** para lógico:

```
(  ) 2000          (  ) "456"          (  ) 1001.0000001
(  ) "RGB"         (  ) 'F'            (  ) "1DF45"
(  ) ';'           (  ) 1.67           (  ) 'I'
(  ) 3.45          (  ) 58             (  ) .F.
(  ) .V.           (  ) '9'            (  ) 0
```

2. Relacione as colunas, caso ocorram relacionamentos:

(1) Teste
(2) bit
(3) Unidade de Controle
(4) Hardware
(5) byte
(6) Sistema Binário
(7) Análise
(8) inteiro
(9) Memória
(10) caractere

( ) Sua função é armazenar os dados e instruções manipuladas pela CPU.
( ) Conjunto de programas que manipula o hardware.
( ) Sistema numérico utilizado nos bytes de memória para representar os números.
( ) Tipo de dados que representa os números inteiros.
( ) Formado pelos circuitos e toda a parte eletrônica do computador.
( ) Tipo de dados que representa os números com ponto decimal.
( ) Pode estar em dois estado possíveis: ligado (1) ou desligado (0).
( ) Tipo de dado que representa seqüências de caracteres.
( ) Contém 8 bits.
( ) Fase do desenvolvimento de um software em que se verifica se o produto realmente funciona como especificado.

3. Mostre como os seguintes dados seriam armazenados em uma memória, de acordo com o modelo ilustrado nas figuras 14 e 15 (especificando o dado armazenado em cada célula e os valores dos bits): **125**, **'A'**, **'CTI'**, **15**.

4. Transforme os seguintes numerais decimais para binário:
a) 1024 b) 77 c) 21

5. Assinale os nomes de variáveis inválidos e descreva o que os tornam inválidos:

```
(  ) -Teste                (  ) GrauEscolaridade
(  ) TeSte                 (  ) Restricao#
(  ) Salario-Familia       (  ) Salario_Mensal
(  ) Numero Matricula      (  ) Teste
(  ) 10Situacao            (  ) DiaSemana
```

6. Declare variáveis para armazenar os seguintes dados:

a) Se uma determinada escola é pública ou não
b) Descrição de um compromisso em uma agenda
c) Número de medalhas de ouro de um país no Pan 2007
d) Preço em reais de um sapato em uma loja
e) Resposta de um aluno em uma questão do vestibular (A, B, C, D ou E)

7. Declare constantes para armazenar os seguintes valores:

a) O número de dias do mês de Agosto (31)
b) O nome de nossa disciplina (Lógica de Programação)

8. O texto "O terreno tem 30 metros de comprimento" é uma informação ou dado? Justifique sua resposta.

3. INTRODUÇÃO AOS ALGORITMOS

Voltamos, neste capítulo, a falar sobre um assunto que já tratamos anteriormente: *algoritmos*. Relembrando, um algoritmo é basicamente a lógica de um programa. É a seqüência ordenada de passos que deve ser seguida para a realização de uma tarefa, garantindo a sua repetibilidade. Em outras palavras, o algoritmo é a seqüência de instruções do programa, escrita de forma independente de linguagem de programação e sem se preocupar com muitos detalhes que nos preocupamos quando fazemos um programa na prática. Simplificando, um algoritmo é o rascunho do programa e um programa é um algoritmo escrito numa forma compreensível pelos computadores.

No Capítulo 1, comparamos a lógica de um programa, ou seja, de um algoritmo, a uma receita de bolo. Esta é uma comparação bem apropriada, pois quando alguém escreve uma receita deseja que outras pessoas a leiam e, através dela, possam chegar ao mesmo bolo. Assim, um algoritmo é uma seqüência de passos de forma que alguém (em nosso caso um computador) possa repetir a tarefa quantas vezes for necessário, da mesma forma e obtendo os mesmos resultados.

Abaixo são mostrados alguns algoritmos que nos permitem repetir algumas tarefas sempre da mesma forma. O Algoritmo 1 nos diz como se fazer um bolo. O Algoritmo 2 nos dá uma seqüência de passos para trocarmos sempre da mesma forma um pneu. Finalmente, o Algoritmo 3 mostra como calcular a média de duas notas de um aluno e diz se ele foi aprovado ou reprovado.

```
Algoritmo Receita de Bolo

   1. Misture os ingredientes
   2. Unte a forma com a manteiga e farinha
   3. Despeje a mistura na forma
   4. Se houver côco ralado então despeje sobre a misture
   5. Leve a forma ao forno
   6. Enquanto não corar deixe a forma no forno
   7. Retire do forno
   8. Deixe esfriar
```

Algoritmo 1: Como fazer um bolo

Diferentes algoritmos podem realizar a mesma tarefa usando um conjunto diferente de instruções em mais ou menos tempo, espaço ou esforço do que outros. Por exemplo, um algoritmo para se vestir pode especificar que você vista primeiro as meias e os sapatos antes de vestir a calça enquanto outro algoritmo especifica que você deve primeiro vestir a calça e depois as meias e os sapatos. Fica claro que o primeiro algoritmo é mais difícil de executar que o segundo apesar de ambos levarem ao mesmo resultado.

Um algoritmo correto deve possuir três qualidades:

1. Cada passo deve ser uma instrução que possa ser realizada, especificada de forma não ambígüa (ou seja, sem mais de uma interpretação)
2. A ordem de execução dos passos deve ser precisamente determinada

3. Deve ter início e fim.

```
Algoritmo Trocar um Pneu

   1. Descer do carro
   2. Pegar o estepe
   3. Pegar o macaco e chave de roda
   4. Colocar o macaco sob o carro
   5. Afrouxar ligeiramente as porcas com a chave de roda
   6. Girar a manivela do macaco para levantar o carro
   7. Retirar os parafusos com auxílio da chave
   8. Retirar o pneu furado
   9. Colocar o estepe
   10.Colocar os parafusos no estepe
   11.Girar a manivela ao contrário para abaixar o carro
   12.Apertar firmemente os parafusos
   13.Retirar o macaco e guardá-lo juntamente com a chave de roda
```

Algoritmo 2: Trocar um pneu furado

```
Algoritmo Cálculo da Média de um Aluno

   1. Peça ao usuário, através do dispositivo de entrada, que informe
      a primeira nota do aluno.
   2. Peça ao usuário, através do dispositivo de entrada, que informe
      a segunda nota do aluno.
   3. Somar as duas notas e dividir o resultado por 2.
   4. Se o resultado do cálculo for maior ou igual a 7.0, então o
      aluno foi aprovado.
   5. Caso contrário, o aluno foi reprovado.
```

Algoritmo 3: Cálculo da média de um aluno e verificação de sua aprovação.

Por exemplo, considere os seguintes algoritmos para contar até 100:

```
Algoritmo Contar até 100 Versão 1

   1. Faça N igual a 0
   2. Some 1 a N
   3. Mostre o valor de N no dispositivo de saída
   4. Volte ao passo 2
```

Algoritmo 4: Contagem até 100 com loop infinito.

```
Algoritmo Contar até 100 Versão 2

   1. Faça N igual a 0
   2. Some 1 a N
   3. Mostre o valor de N no dispositivo de saída
   4. Se N for menor do que 100 volte ao passo 2, caso contrário PARE.
```

Algoritmo 5: Contagem até N correta (sem loop infinito).

A primeira versão não está correta. Ela apresenta o que se chama de *loop infinito*, ou seja, o processamento nunca terminará, pois o algoritmo ficará eternamente executando as instruções 2 e 3. Já a segunda versão fornece uma condição para que o processamento termine normalmente (quando a contagem atingir 100).

Deve ser esclarecido que o desenvolvimento de um algoritmo é uma tarefa não trivial e não determinística; em outras palavras: "não existe um algoritmo que especifique como criar outros algoritmos". Sua criação é baseada puramente na lógica do processamento que se deseja; portanto, vai depender das habilidades intelectuais de cada um.

3.1. Pseudocódigo

Existem diversas formas de representação de algoritmos, mas não há um consenso com relação à melhor delas. Estas formas de representação diferem basicamente quanto ao nível de detalhe oferecido, ou seja, se elas se aproximam mais da descrição em linguagem natural ou mais à linguagem computacional.

Os algoritmos apresentados anteriormente seguem a forma denominada *Descrição Narrativa*, que expressa os algoritmos em linguagem natural. Esta representação é pouco usada na prática porque o uso da linguagem natural muitas vezes dá oportunidade a más interpretações, ambigüidades e imprecisões. Podemos considerar, por exemplo, a instrução "*afrouxar ligeiramente as porcas*" apresentada no Algoritmo 2. Esta instrução está sujeita a interpretações diferentes por pessoas distintas, pois não especifica detalhadamente como realizar o afrouxamento das porcas. Uma instrução mais precisa seria: "*afrouxar cada porca, girando-as 30º no sentido anti-horário utilizando uma chave de roda*".

A forma de representação que utilizaremos é o chamado *Pseudocódigo*, ou *Português Estruturado*. Esta forma de representação é bastante semelhante à forma como os programas serão escritos posteriormente em uma linguagem de programação. Por isso, o pseudocódigo exige mais detalhes na representação do algoritmo do que as outras formas aqui citadas, como definição dos tipos de variáveis, por exemplo. Por ser tão próxima às linguagens de programação, é uma forma de representação que encontra bastante aceitação. Na verdade, esta representação é suficientemente geral para permitir que a tradução de um algoritmo nela representado para uma linguagem de programação específica seja praticamente direta.

A forma geral da representação de um algoritmo na forma de pseudocódigo é a seguinte:

```
Algoritmo <nome_do_algoritmo>;

<declaração_de_constantes>
<declaração_de_variáveis>

Início
     <corpo_do_algoritmo>
Fim.
```

onde:

- **Algoritmo** é uma palavra que indica o início da definição de um algoritmo na forma de pseudocódigo;
- *<nome_do_algoritmo>* é um nome simbólico dado ao algoritmo com a finalidade de distingüi-lo dos demais;
- *<declaração_de_constantes>* e *<declaração_de_variáveis>* consistem em porções opcionais onde são declaradas, respectivamente, as constantes e variáveis usadas no algoritmo (variáveis e constantes não declaradas aqui não podem ser utilizadas no algoritmo);
- **Inicio** e **Fim** são respectivamente as palavras que delimitam o início e término do conjunto de instruções do algoritmo.

Segue abaixo o algoritmo do cálculo da média de um aluno (Algoritmo 3) na forma de pseudocódigo:

```
1  Algoritmo Media;
2  Var
3       N1, N2, Media : real;
4  Inicio
5       Leia N1, N2;
6       Media ← (N1+N2) / 2;
7       Se Media >= 7 Então
8            Escreva "Aprovado";
9       Senão
10           Escreva "Reprovado";
11      Fim Se;
12 Fim.
```

Algoritmo 6: Cálculo da média de um aluno em pseudocódigo.

A linha 1 do algoritmo contém o início da definição do mesmo, onde é definido o seu nome ("Media"). Logo a seguir, encontramos a definição das variáveis utilizadas no algoritmo, da mesma forma que vimos no capítulo anterior. Neste algoritmo foram definidas três variáveis do tipo real (N1, N2 e Media). A instrução Leia na linha 5 corresponde a instruções 1 e 2 do algoritmo em

descrição narrativa. A instrução na linha 6 corresponde ao cálculo da média apresentado no passo 3 do algoritmo original. Os passos 4 e 5 do algoritmo em descrição narrativa correspondem às linhas 7 a 11 do Algoritmo 6.

3.2. Instruções Primitivas

Instruções primitivas são comandos básicos utilizados em algoritmos para executar tarefas essenciais como entrada e saída de dados (comunicação com o usuário através de dispositivos periféricos), e movimentação dos mesmos na memória. Estes tipos de instrução estão presentes em todas as linguagens de programação. De fato, um programa que não utiliza nenhuma instrução primitiva – como as que serão definidas neste capítulo – é incapaz de se comunicar com o mundo exterior e, portanto, não tem utilidade nenhuma.

Antes de passarmos à descrição das instruções primitivas, é necessária a definição de alguns termos que serão utilizados mais à frente:

- *sintaxe* é a forma como os comandos devem ser escritos, a fim de que possam ser entendidos pelo tradutor de programas (compilador ou interpretador). A violação das regras sintáticas é considerada um erro sujeito à pena de não-reconhecimento do comando por parte do tradutor;
- *semântica* é o significado, ou seja, o conjunto de ações que serão exercidas pelo computador durante a execução do referido comando.

Daqui para frente, todos os comandos serão apresentados através de sua sintaxe e de sua semântica, isto é, a forma em que devem ser escritos e as ações que executam.

3.2.1. Instrução Primitiva de Atribuição

A *instrução primitiva de atribuição*, ou simplesmente *atribuição*, é a principal maneira de armazenar um dado em uma variável. Sua sintaxe é:

```
<nome_de_variável> ← <valor>;
```

O valor expresso ao lado direito do comando é armazenado na variável informada à esquerda do comando (<nome_de_variável>). Este valor pode ser o nome de uma constante ou até outra variável, de onde o valor a ser armazenado pode ser obtido.

Uma implicação bastante séria, para qual a atenção deve ser dirigida, é a necessidade da compatibilidade entre o tipo de dado do valor e o tipo de dado da variável, no sentido em que esta deve ser capaz de armazenar o valor. Por exemplo, se colocamos ao lado direito da atribuição um valor lógico, então a variável deve ser também do tipo lógico. Uma exceção é o caso em que a variável é do tipo real e o valor é do tipo inteiro. Nesta situação, o dado do tipo inteiro é convertido para o tipo real e posteriormente armazenado na variável.

O Algoritmo 7 abaixo mostra um exemplo onde algumas atribuições são feitas: os valores 5.0 e 10 (valor da constante Teste) são atribuídos às variáveis PrecoUnit e Quant, respectivamente; PrecoTot conterá o valor da variável PrecoUnit no momento da atribuição, ou seja, 5.0.

```
Algoritmo PrecoTotal1;

const
      Teste : inteiro = 10;

var
      PrecoUnit, PrecoTot : real;
      Quant : inteiro;

inicio
      PrecoUnit ← 5.0;
      Quant ← Teste;
      PrecoTot ← PrecoUnit;
fim.
```

Algoritmo 7: Exemplo mostrando o uso de atribuições em algoritmos.

Sempre utilizaremos uma instrução de atribuição quando escreveríamos, em descrição narrativa, algo como "armazene o valor X na variável Y", "copie o valor da variável A na variável B" e etc.

3.2.2. Instrução Primitiva de Saída de Dados

Como vimos nos exemplos e exercícios de algoritmos em descrição narrativa, é necessária uma forma de mostrar dados ao usuário, como resposta à execução de um programa. As instruções primitivas de saída dados são o meio pelo qual informações contidas na memória dos computadores podem ser mostradas nos dispositivos de saída, para que o usuário possa visualizá-las.

A sintaxe para esta instrução é:

escreva <*literal*>

ou

escreva <*variável*>

O literal a ser informado na instrução é simplesmente um dado do tipo literal delimitado por aspas duplas ou, como veremos mais adiante, uma expressão que resulte em um literal. Caso se utilize uma variável, o nome desta deve ser informado no comando. A semântica da instrução primitiva de saída de dados é muito simples: o valor literal ou da variável é enviado para o dispositivo de saída.

O Algoritmo 8 mostra o Algoritmo 7 modificado, com o acréscimo de uma instrução "escreva" para mostrar o conteúdo da variável PrecoTot ao final da execução do algoritmo (o valor 5.0 será mostrado). Já o Algoritmo 9 mostra no dispositivo de saída de dados um retângulo montado com asteriscos contendo o texto "Lógica de Programação".

```
Algoritmo PrecoTotal2;

const
      Teste : inteiro = 10;

var
      PrecoUnit, PrecoTot : real;
      Quant : inteiro;

inicio
      PrecoUnit ← 5.0;
      Quant ← Teste;
      PrecoTot ← PrecoUnit;
      escreva PrecoTot;
fim.
```

Algoritmo 8: Algoritmo mostrando a utilização da instrução "escreva" para mostrar o conteúdo de uma variável.

```
Algoritmo MostraQuadrado;
inicio
      escreva "*************************";
      escreva "* Lógica de Programação *";
      escreva "*************************";
fim.
```

Algoritmo 9: Um algoritmo que mostra a mensagem "Lógica de Programação" dentro de um retângulo de asteriscos.

Note que nenhum dispositivo de saída é especificado na instrução "escreva". Nos algoritmos não nos preocuparemos com detalhes desse nível, apenas de que o dado desejado será mostrado de alguma maneira ao usuário. Na maioria das linguagens de programação existem instruções desse tipo, mas os dados são mostrados em um dispositivo de saída padrão, geralmente o monitor de vídeo.

3.2.3. Instrução Primitiva de Entrada de Dados

Já vimos como atribuir valores a variáveis e como mostrar dados para o usuário. Mas como obter dados dos usuários, como já fizemos em algoritmos utilizando descrição narrativa? A instrução primitiva de entrada de dados foi criada para suprir esta necessidade.

Sua sintaxe é:

leia <*lista_de_variáveis*>

A *lista_de_variáveis* é um conjunto de um ou mais nomes de variáveis, separados por vírgulas.

A semântica da instrução de entrada (ou leitura) de dados é, de certa forma, inversa à da instrução de escrita: os dados são fornecidos ao computador por meio de um dispositivo de entrada e armazenados nas variáveis cujos nomes aparecem na `lista_de_variáveis`.

Podemos modificar o Algoritmo 8 para solicitarmos ao usuário que digite um valor para a

variável PrecoUnit. O Algoritmo 10 mostra como ficaria esta modificação. A instrução leia esperará até que o usuário informe, através do dispositivo de entrada, um valor real. Este valor será armazenado dentro da variável PrecoUnit.

```
Algoritmo PrecoTotal3;

const
      Teste : inteiro = 10;

var
      PrecoUnit, PrecoTot : real;
      Quant : inteiro;

inicio
      leia PrecoUnit;
      Quant ← Teste;
      PrecoTot ← PrecoUnit;
      escreva PrecoTot;
fim.
```

Algoritmo 10: Algoritmo mostrando o uso da instrução "leia" para preencher a variável PrecoUnit com um valor real digitado pelo usuário.

O Algoritmo 11 é um algoritmo que lê do dispositivo de entrada um nome e mostra este nome novamente na tela após a mensagem “Bem-vindo”. Duas instruções “escreva” são utilizadas neste exemplo. Como veremos mais adiante, podemos, através de uma expressão, mostrar esta informação em uma única instrução “escreva”.

```
Algoritmo BemVindo;

var
      nome : literal;

inicio
      leia nome;
      escreva “Bem-vindo ”;
      escreva nome;
fim.
```

Algoritmo 11: Algoritmo que lê um nome da entrada padrão e mostra uma mensagem de boas vindas.

3.3. Delimitadores de Instruções

Você já deve ter notado que todas as instruções dentro do corpo do algoritmo terminam com um ponto-e-vírgula (;) ao final da linha. Este ponto-e-vírgula é obrigatório, e serve para marcar o fim de uma instrução. Várias instruções podem ser colocadas na mesma linha, mas deve haver um ponto-e-vírgula no final de cada uma. Esquecer um ponto-e-vírgula é um erro, que seria acusado por um compilador ou interpretador durante o processo de tradução. O Algoritmo 12 mostra o algoritmo Algoritmo 11 modificado, contendo todas as três instruções em uma mesma linha.

```
Algoritmo BemVindo2;

var
     nome : literal;

inicio
     leia nome; escreva "Bem-vindo "; escreva nome;
fim.
```

Algoritmo 12: Algoritmo contendo todas as instruções na mesma linha.

3.4. Exercícios

1. Escreva um algoritmo que mostre a mensagem "Alô mundo!" na tela.

2. Escreva algoritmos para mostrar as seguintes figuras geométricas na tela, utilizando asteriscos:

```
a) ****      b) *         c) *****     d) *****
   ****         **           *            ****
   ****         ***          ***          ***
                ****         *            **
                             *****        *
```

3. Marque (V)erdadeiro ou (F)also para cada afirmativa abaixo, sobre o algoritmo mostrado.

```
1  Algoritmo Exercicio3;
2  const
3      valor : inteiro = 123;
4  var
5      x, y : inteiro;
6  inicio
7      x ← valor;
8      y ← 9;
9      escreva y;
10     x ← 50;
11     escreva x;
12     y ← 0;
13 fim.
```

( ) Os primeiro valor a ser mostrado na tela será 0 (zero).

( ) Na linha 7 é atribuído o valor 123 para a variável x.

( ) O segundo valor a ser mostrado será 50.

( ) A atribuição efetuada na linha 12 é inválida.

4. O algoritmo abaixo possui alguns erros. Que erros são esses?

```
1 Algoritmo Exercicio6;
2 var
        a, b : inteiro;
3 inicio
4       a ← 5.0;
5       b ← 6;
6       leia x;
7       escreva a;
8 fim.
```

5. Escreva um algoritmo que declare duas variáveis inteiras, A e B. O algoritmo deve atribuir à variável A o valor 3 e à variável B o valor 5. Após isso, o algoritmo deve trocar os valores das variáveis, de forma que A seja 5 e B seja 3 (não deve ser feita uma atribuição direta de A para 5 e B para 3).

6. Escreva um algoritmo que leia dois caracteres e mostre os dois invertidos na saída. Ou seja, se o usuário digitar A e C, deve ser mostrado na saída C e depois A.

7. Modifique o Algoritmo 11 para que mostre a mensagem de boas vindas em inglês (Bem-vindo em inglês é *Welcome*).

8. Escreva um algoritmo de boas vindas que solicite ao usuário que digite seu nome e sobrenome e logo após mostre a mensagem "Bem-vindo Sr(a) " seguida do sobrenome da pessoa.

9. Escreva um algoritmo que mostre o nome do seu curso e o nome da disciplina. O nome do curso deve ser armazenado em uma constante e o nome da disciplina em uma variável. As instruções de saída de dados devem buscar os valores na memória.

4. Introdução à Linguagem Java

No capítulo 1, vimos que os programas de computador são escritos por pessoas chamadas programadores. No capítulo anterior, vimos também que a lógica de um programa, ou seja, a seqüência de instruções do que ele deve fazer, é expressa através de um algoritmo. Este algoritmo torna-se um programa de fato quando é transcrito em uma linguagem de programação e, assim, pode ser compilado ou interpretado.

Centenas de linguagens de programação de alto nível foram desenvolvidas, mas somente algumas delas alcançaram sucesso e foram largamente utilizadas. Dentre algumas delas podemos citar:

- *FORTRAN* (*FORmula TRANslator*), primeira linguagem de alto nível, voltada para o desenvolvimento de aplicações que requerem cálculos matemáticos complexos;
- *COBOL* (*COmmon Business Oriented Language*), criada principalmente para o desenvolvimento de aplicativos comerciais;
- *Pascal*, linguagem de uso geral, utilizada principalmente no ensino acadêmico;
- *Basic*, uma linguagem simples voltada para iniciantes em programação;
- *C*, linguagem de uso geral largamente utilizada hoje para escrever sistemas operacionais, compiladores e programas que acessem diretamente o hardware;
- *C++*, um aprimoramento da linguagem C que passou a utilizar conceitos de orientação a objetos.

A linguagem que iremos utilizar em nosso curso, na implementação de nossos algoritmos, será a linguagem *Java*. Esta linguagem é amplamente utilizada hoje, principalmente no desenvolvimento de aplicações para a Internet.

Java surgiu a partir de um projeto da empresa norte-americana *Sun Microsystems*, em 1991, cujo codinome era *Green*. Este projeto visava o desenvolvimento de dispositivos eletrônicos inteligentes destinados ao consumidor final. O projeto resultou no desenvolvimento de uma linguagem baseada em C e C++ chamada *Oak* (carvalho). Descobriu-se mais tarde que já havia uma linguagem chamada Oak e ela foi rebatizada de Java (devido ao nome da cidade de origem de um tipo de café importado[1]).

O projeto Green atravessou dificuldades e estava em risco de ser cancelado. Mas em 1993, a Internet explodiu em popularidade e os projetistas da Sun viram o imediato potencial da utilização de Java para criar páginas na rede com o chamado *conteúdo dinâmico*. Isso deu nova vida ao projeto.

Java é hoje utilizada para criar páginas na Internet com conteúdo dinâmico e interativo, para desenvolver aplicativos corporativos de grande porte, e também no desenvolvimento de programas para dispositivos como telefones celulares, computadores de mão e a TV digital interativa.

1 Os projetistas da linguagem estavam em uma cafeteria quando tiveram a brilhante idéia.

4.1. Estrutura Básica de um Programa Java

Os programas em Java consistem em partes chamadas *classes*. Estas consistem, por sua vez, em partes chamadas *métodos*, que realizam tarefas e retornam os dados resultantes ao completarem seu trabalho. O desenvolvimento de programas em Java consiste na escrita de classes e seus respectivos métodos.

O Programa 1 é um aplicativo que exibe uma linha de texto contendo a mensagem "Bem-vindo ao mundo Java!". Esse pequeno programa, apesar de simples, ilustra vários recursos importantes da linguagem Java. Consideremos cada linha em detalhe (os números das linhas encontram-se ao lado de cada uma e não fazem parte do programa em si).

```
1     // Programa 1: Bemvindo1.java
2     // Meu primeiro programa em Java
3
4     public class Bemvindo1 {
5
6           // o método main inicia a execução do aplicativo Java
7           public static void main(String[] args)
8           {
9                 System.out.println("Bem-vindo ao mundo Java!");
10          } // fim do método main
11
12    } // fim da classe Bemvindo1
```

Programa 1: Programa que mostra uma mensagem de boas vindas na tela.

A linha 1 inicia com `//`, indicando que o restante da linha é um *comentário*. Os programadores inserem comentários no código-fonte de seus programas para descrever o que determinadas linhas fazem ou para descrever aspectos gerais do programa. Os comentários são úteis para tornar o código mais claro, ajudando outras pessoas a ler e entender o programa. O compilador ignora comentários, ou seja, os comentários não são instruções que realizam alguma ação quando o programa é executado.

O comentário que inicia com `//` é chamado *comentário de uma única linha*, pois o comentário inicia logo após as barras e termina ao final da linha. Se desejamos um comentário de múltiplas linhas, devemos iniciar cada uma com `//`. Observe que um comentário pode iniciar no meio de uma linha e continuar até o final dessa linha (veja as linhas 10 e 12 do Programa 1, por exemplo).

Os comentários de múltiplas linhas podem ser escritos de outra maneira. Por exemplo, os comentários escritos nas linhas 1 e 2 poderiam ser reescritos da seguinte forma:

```
/* Programa 1: Bemvindo1.java
   Meu primeiro programa em Java */
```

Este tipo de comentário inicia com o delimitador `/*` e termina com o delimitador `*/`. Todo o texto entre os delimitadores é ignorado pelo compilador.

A linha 2 apresenta um comentário de uma única linha que descreve o propósito do programa. É conveniente iniciar um programa com um comentário que descreva seu propósito.

A linha 3 é simplesmente uma linha em branco. Os programadores usam linhas em branco e caracteres de espaçamento para tornar os programas mais fáceis de ler. Esses caracteres são ignorados pelo compilador.

A linha 4 inicia a definição da classe Bemvindo1. Cada programa Java consiste em pelo menos uma classe definida pelo programador. A palavra-chave *class* inicia uma definição de classe em Java e é imediatamente seguida pelo nome da classe (em nosso exemplo, Bemvindo1). As *palavras-chave* (ou *palavras reservadas*) são reservadas para uso exclusivo de Java e não devem ser utilizadas em nomes de classes, variáveis e constantes e são sempre escritas com letras minúsculas.

Aliás, esse é um ponto que devemos tomar cuidado. A linguagem Java diferencia letras maiúsculas de minúsculas. Assim, "Class" será considerado diferente de "class".

Por convenção, todos os nomes de classe em Java iniciam com uma letra maiúscula e têm uma letra maiúscula para cada palavra no nome da classe (por exemplo, ExemploDeNomeDeClasse). Além disso, os nomes de classes seguem as mesmas regras utilizadas para nomenclatura de variáveis em algoritmos (ver Capítulo 2).

A classe que definimos inicia com a palavra-chave *public*. Discutiremos o significado dessa palavra mais adiante em nosso curso. Por enquanto devemos saber que sempre devemos utilizá-la na definição de nossas classes.

Um programa Java deve ser salvo em um arquivo que tenha o mesmo nome da classe, seguido pela extensão ".java". Para nosso pequeno programa, o nome do arquivo é "Bemvindo1.java". Arquivos que não seguirem essa regra não poderão ser compilados.

Ao final da linha 4, encontramos uma chave esquerda, {, que indica o início do *corpo* da classe. A chave direita correspondente (linha 12), }, deve ser inserida ao final da definição de cada classe que escrevermos.

As linhas 6 a 10 estão recuadas, ou seja, mais à direita do que as demais. Esse recuo é uma convenção de espaçamento utilizada para facilitar a leitura e entendimento do programa.

A linha 5 é uma linha em branco, inserida para melhorar a legibilidade do programa. A linha 6 é um comentário de uma linha que indica a finalidade das linhas 7 a 10 do programa.

A linha 7 é obrigatória em todo programa Java. A execução de um programa em Java começa pelas instruções contidas no bloco *main*. Os parênteses depois da palavra *main* indicam ser ele um bloco de instruções denominado *método*. Como já mencionamos, as classes Java normalmente contém um ou mais métodos. Para uma classe de programa Java, como em nosso exemplo, exatamente um desses métodos deve ser chamado de *main* e deve ser escrito como mostrado na linha 7. Veremos mais detalhes sobre métodos mais tarde, e entenderemos o significado das palavras *public*, *static* e *void*. Por enquanto, simplesmente copie a primeira linha do método *main* em cada um de seus programas Java.

A chave esquerda, {, na linha 8, inicia o corpo da definição do método *main*. A chave direita correspondente, }, deve terminar o corpo da definição do método(linha 10). Essas chaves correspondem às palavras "inicio" e "fim." nos algoritmos em pseudocódigo. Observe que a linha no corpo do método (linha 9) está recuada entre essas chaves. É dentro do método *main* que ficarão as instruções dos programas que escreveremos nos próximos capítulos.

A instrução `System.out.println` equivale a um comando "escreva" do pseudocódigo.

Veremos mais sobre esta instrução nas próximas seções. Note que a linha termina com um ponto-e-vírgula (;). Assim como nos algoritmos, cada instrução em um programa Java deve terminar com um ponto-e-vírgula.

Resumindo, a estrutura básica de um programa Java é mostrada na Figura 17. Ao lado, é mostrada também a estrutura de um algoritmo em pseudocódigo, para comparação.

Algoritmo *nome*; **inicio** *instruções* **fim.**	**public class** *nome* { **public static void** main(String[] args) { *instruções* } }

Figura 17: Estrutura básica resumida de um programa Java e seu equivalente em pseudocódigo.

4.2. Compilando e Executando um Programa Java

Os programas em Java podem ser escritos em qualquer editor de textos, como o *WordPad* no *Windows* ou *Kate* em *Linux*. É só escrever o código e salvá-lo como um arquivo de texto comum, com o nome apropriado (como descrito na seção anterior). Em nosso exemplo, o arquivo deve ser salvo com o nome de *Bemvindo1.java*.

Para compilar o programa, abrimos uma janela de comando (como o *Prompt de Comando* no Windows ou um *terminal shell* no Linux), mudamos para o diretório onde o programa é armazenado e digitamos

```
javac Bemvindo1.java
```

O programa javac é o compilador Java. É ele quem transforma o código-fonte, escrito em Java, em um código que pode ser executado no computador.

Se o programa não contiver erros de sintaxe, o comando precedente criará um novo arquivo executável, chamado *Bemvindo1.class*. Este arquivo gerado pelo compilador não contém o código de máquina equivalente ao programa, mas um outro tipo de codificação intermediária chamado de *bytecode*. O bytecode não pode ser executado diretamente pela CPU do computador, mas precisa de um interpretador chamado de *máquina virtual Java*. Os programas em Java seguem esse modelo diferenciado devido ao desígnio original da linguagem, que era a de executar o mesmo programa em diferentes tipos de computadores, sem a necessidade de recompilação, mas mais rápido do que a utilização direta de um interpretador.

Logo, para executarmos nosso programa, precisamos do auxílio da máquina virtual Java. Para isso, basta digitarmos na linha de comando

```
java Bemvindo1
```

que executa o interpretador de bytecodes e carrega o arquivo ".class" para a classe Bemvindo1. Observe que a extensão ".class" foi omitida no programa precedente; caso contrário o interpretador não executaria o programa. O interpretador chama automaticamente o método *main*. Em seguida, a instrução na linha 9 exibe a mensagem "Bem-vindo ao mundo Java!". A seqüência de comandos descrita é mostrada em uma janela de comando do Windows, na Figura 18.

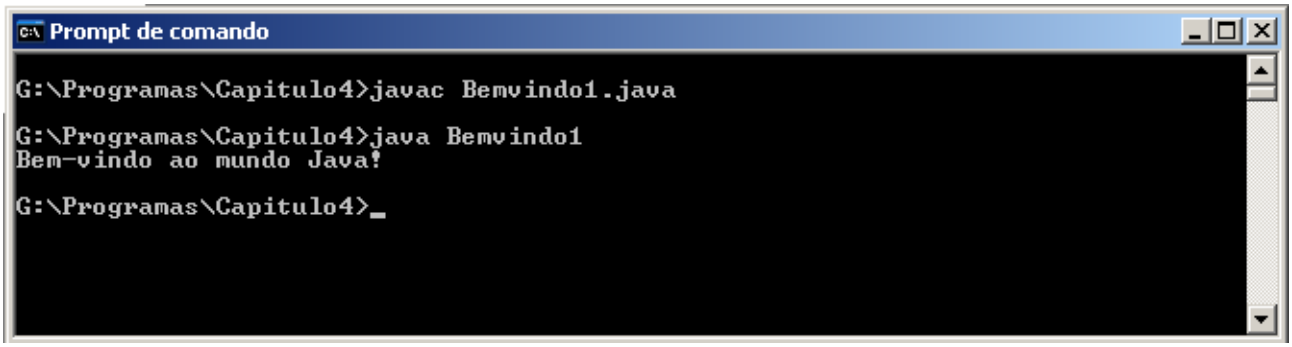


Figura 18: Compilação e execução do programa Bemvindo1.java

4.3. Comandos de Saída de Dados em Java

Em algoritmos utilizando pseudocódigo, temos apenas uma opção para saída de dados: o comando "escreva". Já em Java temos uma dezena de opções, dependendo do tipo de saída que desejamos. Veremos neste capítulo três opções.

A primeira já foi mostrada na linha 9 do Programa 1. `System.out` é conhecido como *objeto de saída padrão*. Ele permite exibir literais e outros tipos de dados na janela de comando a partir da qual o programa é executado. No Windows 95/98/ME, a janela de comando é o *Prompt do MS-DOS*. No Windows NT/2000/XP, a janela de comando é o *Prompt de Comando* (cmd.exe). Em Linux e outros sistemas derivados do Unix, a janela de comando normalmente é chamada de *shell*, *terminal shell* ou apenas *terminal*.

O método `System.out.println` exibe uma linha de texto na janela de comando, da mesma forma que o comando escreva nos algoritmos. Quando System.out.println completa sua tarefa, automaticamente posiciona o *cursor de saída* (o lugar onde o próximo caractere será exibido na tela, geralmente indicado com um traço piscante) no início da próxima linha na janela de comando (é como se a instrução pressionasse a tecla *Enter* após exibir a mensagem).

```
1     // Programa 2: Bemvindo2.java
2     // Mostra uma linha de texto utilizando múltiplas instruções.
3
4     public class Bemvindo2 {
5
6           // o método main inicia a execução do aplicativo Java
7           public static void main(String[] args)
8           {
9                 System.out.print("Bem-vindo ao ");
10                System.out.println("mundo Java!");
10          } // fim do método main
11
12    } // fim da classe Bemvindo2
```

Programa 2: Modificação do programa Bemvindo1 mostrando a utilização do método System.out.print.

Uma variante do método println de System.out é o método *print*. Ele exibe a mensagem de texto da mesma forma que println, mas não posiciona o cursor na próxima linha ao final. Ou seja, o

cursor de saída fica posicionado logo após a mensagem, na mesma linha. O Programa 2 mostra o Programa 1 modificado. Nesta nova versão, o programa utiliza uma combinação de instruções System.out.print e System.out.println (linhas 9 e 10) para obter o mesmo resultado do Programa 1. Cada instrução print ou println retoma a exibição do texto a partir de onde o último print ou println parou de exibir os caracteres.

É possível exibir, em uma única instrução de saída várias linhas de texto. Para isso utilizamos caractere especial de nova linha, representado pela seqüência `\n`. Quando esta combinação é encontrada, o cursor é posicionado em uma nova linha e a exibição prossegue a partir do próximo caractere. O Programa 3 mostra a mensagem "Bem-vindo ao mundo Java!" em várias linhas, como mostrado na Figura 19.

```
1     // Programa 3: Bemvindo3.java
2     // Mostra várias linhas de texto com uma única instrução.
3
4     public class Bemvindo3 {
5
6           // o método main inicia a execução do aplicativo Java
7           public static void main(String[] args)
8           {
9                 System.out.println("Bem-vindo\nao\nmundo\nJava!");
10          } // fim do método main
11
12    } // fim da classe Bemvindo3
```

Programa 3: Programa Bemvindo1 modificado para mostrar a mensagem em várias linhas, utilizando uma única instrução de saída de dados.

```
Bem-vindo
ao
mundo
Java!
```

Figura 19: Saída exibida pelo programa Bemvindo3.

Existe ainda outra forma de mostrar mensagens ao usuário, utilizando *caixas de diálogo*. Muitos programas que você usa utilizam esse recurso. A classe *JOptionPane,* da linguagem Java, oferece caixas de diálogo predefinidas que permitem aos programas exibir mensagens simples ao usuário.

A tecnologia Java oferece um rico conjunto de classes predefinidas, as quais os programadores podem utilizar em vez de "reinventar a roda". Essas diversas classes predefinidas são agrupadas em categorias de classes relacionadas chamadas *pacotes*. O conjunto de pacotes disponíveis é conhecido coletivamente como *API* (*Applications Programming Interface – Interface de Programação de Aplicativos*). A classe JOptionPane que utilizaremos à seguir pertence a um destes pacotes predefinidos, chamado *javax.swing*.

```
1     // Programa 4: Bemvindo4.java
2     // Mostra uma mensagem de texto utilizando uma caixa de diálogo.
3
4     // Importa a classe JOptionPane do pacote javax.swing
5     import javax.swing.JOptionPane;
6
7     public class Bemvindo4 {
8
9           // o método main inicia a execução do aplicativo Java
10          public static void main(String[] args)
11          {
12                JOptionPane.showMessageDialog(
13                   null, "Bem-vindo\nao\nmundo\nJava!");
14
15                System.exit( 0 );     // termina o aplicativo
16          } // fim do método main
17
18    } // fim da classe Bemvindo4
```

Programa 4: Programa Bemvindo3 modificado para utilizar saída em caixa de diálogo.

O Programa 4 mostra a mesma mensagem mostrada pelo Programa 3, mas utilizando uma caixa de diálogo. Vejamos algumas das modificações efetuadas.

A linha 5 é uma instrução *import*. Utilizamos instruções *import* para dizer ao compilador onde localizar classes da API Java que estamos utilizando em nosso programa. Na instrução *import* que utilizamos em nosso exemplo, dizemos ao compilador para carregar a classe *JOptionPane* do pacote *javax.swing*. Esse pacote contém muitas classes para a definição de *interfaces gráficas com o usuário* (*GUIs – Graphical User Interfaces*). Essas classes implementam componentes de tela, como janelas, botões, menus, caixas de texto, etc.

As linhas 12 e 13 do método *main* contém uma chamada para o método *showMessageDialog* da classe JOptionPane. Este método necessita de dois *argumentos*, ou seja, precisamos informar a ele, entre parênteses, uma lista de dois dados para que ele funcione. Note que os dados entre parênteses são separados por vírgulas. Até discutirmos a classe JOptionPane em mais detalhes, o primeiro argumento será sempre *null*. Este primeiro argumento serve para posicionar a janela a ser exibida na tela. Quando o valor deste argumento é *null*, a janela aparece no centro da tela do computador. O segundo argumento é o dado literal a ser exibido.

Lembre-se que uma instrução Java termina com um ponto-e-vírgula (;). Logo, as linhas 12 e 13 representam uma única instrução, dividida em duas linhas. A linguagem Java permite que uma instrução seja dividida em várias linhas, desde que esta divisão não seja feita no meio de um dado do tipo literal e nem em algum nome de variável, constante ou outra entidade do programa (como classes, métodos e outras que veremos mais adiante).

A execução da instrução nas linhas 12 e 13 mostra a janela mostrada na Figura 20. A barra de título contém o literal *Message* indicando que o contéudo da janela é uma mensagem para o usuário. A janela exibida pode ser fechada clicando-se com o botão esquerdo do mouse sobre o botão *OK*.

Por fim, a linha 15 utiliza o método *exit* da classe *System* para terminar o programa. Isto é necessário quando escrevemos programas que utilizam interface gráfica. A classe *System* pertence

ao pacote *java.lang*. Por padrão, o pacote *java.lang* é importado em todos os programas Java. Por isso não escrevemos um comando import para a classe *System*.

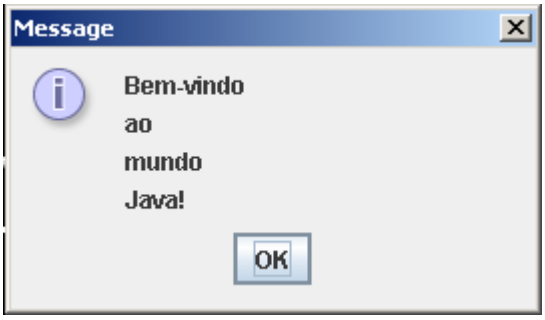


Figura 20: Saída gerada pelo método showMessageDialog no programa Bemvindo4.

4.4. Tipos de Dados em Java

A Tabela 4 mostra a lista de tipos de dados de Java, equivalente aos tipos que já vimos nos algoritmos. A primeira coluna mostra os tipos que estudamos nos algoritmos. A segunda coluna mostra os tipos equivalentes em Java (note que alguns tipos possuem mais de um equivalente). A terceira coluna mostra o espaço de memória ocupado por valores dos tipos indicados em Java. Na prática, dados ocupam espaços determinados de memória, o que restringe o conjunto de valores representados por cada tipo. A última coluna mostra exatamente o conjunto de valores representado por cada tipo.

Note que podemos utilizar quatro tipos diferentes para representar um dado inteiro. Mas cuidado: cada tipo possui um conjunto limitado de valores possíveis. Por exemplo, o número 200 não pode ser representado com o tipo *byte*, mas pode ser representado pelos tipos *short*, *int* e *long*. Em geral, o tipo *int* é mais utilizado para guardar valores inteiros e o tipo *double* para valores reais, mas isso não é uma regra geral. A escolha do tipo mais adequado vai depender do bom senso do programador.

Valores do tipo *char* devem ser escritos entre aspas simples, como nos algoritmos. Assim também, os valores do tipo *String*, devem ser escritos entre aspas duplas. Os valores dos tipos *float* e *double* devem utilizar o ponto decimal, como trabalhamos nos algoritmos.

4.5. Variáveis e Constantes em Java

Nos algoritmos em pseudocódigo, as variáveis são declaradas em uma área especial iniciada pela palavra *var*. Em Java, as variáveis podem ser declaradas em qualquer lugar do programa, desde que dentro do método *main*. Devemos tomar cuidado com um detalhe: a variável só existe a partir do momento em que é declarada. É um erro tentar usar uma variável em uma instrução antes que esta tenha sido declarada.

Tipos em Algoritmos	***Tipos em Java***	***Tamanho em bits***	***Valores***
inteiro	byte short int long	8 16 32 64	-128 a +127 (-2^7 a 2^7-1) -32768 a +32767 (-2^{15} a $2^{15}-1$) -2^{31} a $2^{31}-1$ -2^{63} a $2^{63}-1$
real	float double	32 64	*Intervalor negativo:* -3,4028234663852886e+38 a -1,40129846432481707e-45 *Intervalo positivo:* 1,40129846432481707e-45 a 3,4028234663852886e+38 *Intervalo negativo:* -1,7976931348623157e+308 a -4,94065645841246544e-324 *Intervalo positivo:* 4,94065645841246544e-324 a 1,7976931348623157e+308
lógico	boolean	8	*true* ou *false*
caractere	char	16	Conjunto de caracteres Unicode ISO
literal	String	variável	

Tabela 4: Tipos de dados em Java e em algoritmos.

Para se declarar uma variável em Java, basta escrever seu tipo seguido do nome da variável, como mostrado abaixo:

```
int a, b;
char letra;
String texto, msg;
boolean flag;
```

No exemplo acima são declaradas 6 variáveis: “a” e “b” do tipo int, “letra” do tipo char, “texto” e “msg” do tipo String e “flag” do tipo boolean. Estas declarações equivalem ao seguinte, em pseudocódigo:

```
var
     a, b       : inteiro;
     letra      : caractere;
     texto, msg : literal;
     flag       : lógico;
```

Como mostrado, podemos declarar diversas variáveis de um mesmo tipo em uma única instrução de declaração, separando seus nomes por vírgulas. A declaração de variáveis deve sempre terminar com um ponto-e-vírgula.

As constantes podem ser declaradas da mesma forma das variáveis, mas a declaração deve ser precedida da palavra *final*. Para declararmos, for exemplo, uma constante inteira com o valor 5, podemos escrever o seguinte código:

```
final int numero = 5;
```

Deve ser atribuído um valor para a constante na sua declaração, assim como nos algoritmos (após o sinal de igual). Em pseudocódigo, a mesma constante seria declarada dentro da seção const da seguinte forma:

```
const numero : inteiro = 5;
```

As regras para nomenclatura de variáveis e constantes em Java são as mesmas que vimos para os algoritmos.

4.6. Instrução de Atribuição em Java

Vimos no capítulo anterior que, para armazenarmos valores em variáveis, utilizamos a instrução primitiva de atribuição, simbolizada por ←. Em Java existe uma instrução semelhante de atribuição, simbolizada pelo sinal de igual (=). As linhas abaixo representam instruções que atribuem os valores 5, "Teste" e false para as variáveis *num*, *texto* e *logico*, respectivamente:

```
num = 5;              // Equivalente a num ← 5;
texto = "Teste";      // Equivalente a texto ← "Teste";
logico = false;       // Equivalente a logico ← .F.;
```

O Programa 5 é uma transcrição do Algoritmo 7 para Java. Note a declaração das variáveis no meio das instruções do programa. Como dissemos isso é perfeitamente legal em Java. A variável precoTot, por exemplo, só foi ser declarada na linha anterior em que foi usada. Isso é uma prática comum em Java. Entretanto, a variável PrecoTot só pode ser usada após a sua declaração, ou seja, da linha 18 em diante. Usar a variável em uma linha anterior seria um erro.

A linguagem Java permite ainda que se atribue um valor a uma variável no momento de sua declaração. Por exemplo, poderíamos reescrever as linhas 11 a 18 do Programa 5 da seguinte forma:

```
double precoUnit = 10;
int quant = teste;
double precoTot = precoUnit;
```

```
1     // Programa 5: PrecoTotal1.java
2     // Exemplo utilizando instruções de atribuição.
3
4     public class PrecoTotal1 {
5
6           // o método main inicia a execução do aplicativo Java
7           public static void main(String[] args)
8           {
9                 final int teste = 10;
10
11                double precoUnit;
12                precoUnit = 5.0;
13
14                int quant;
15                quant = teste;
16
17                double precoTot;
18                precoTot = precoUnit;
19
20          } // fim do método main
21
22    } // fim da classe PrecoTotal1
```

Programa 5: Algoritmo 7 transcrito para Java.

4.7. Comandos de Entrada de Dados em Java

Nos algoritmos que vimos no capítulo anterior utilizamos a instrução "leia" para realizar entrada de dados, ou seja, para solicitar ao usuário que digitasse algum dado necessário para o programa. Veremos nesta seção duas formas de realizar entrada de dados em Java: uma pelo terminal de comandos e outra por caixa de diálogo.

A primeira, utilizando o terminal de comandos, faz uso da classe *Scanner*. Esta classe pertence ao pacote *java.util*. Logo, é necessário importar a classe para se fazer uso dela no programa. O Programa 6 mostra um exemplo, implementando o Algoritmo 11 em Java, que utiliza a classe Scanner para realizar a leitura do nome do usuário e mostrar uma mensagem de boas-vindas.

Vamos analisar algumas linhas desse programa. A linha 5 contém a instrução para importação da classe *Scanner*. A linha 12 de nosso programa contém a declaração de uma variável do tipo *Scanner,* chamada *entrada.* Veremos mais adiante, que classes em Java podem ser utilizadas como tipos de dados (assim como *int*, *double*, etc).

Na linha 13, atribuímos à variável *entrada* um novo *objeto* do tipo *Scanner*. Veremos mais sobre objetos mais tarde. Podemos visualizar objetos, neste caso, da seguinte forma: a classe *Scanner* define a estrutura que todos os objetos devem ter (é como uma fôrma de bolo; todos os bolos feitos nessa fôrma terão a mesma forma); os objetos por sua vez são entidades que seguem esse padrão, mas que são diferentes uns dos outros (a partir de uma fôrma redonda, que é a classe, podemos fazer um bolo de chocolate e outro de fubá, que são dois objetos diferentes, mas com a mesma forma). Na linha 13, então, criamos um novo bolo a partir da forma *Scanner*. O texto entre parênteses, *System.in*, informa que estamos criando um novo objeto *Scanner* para leitura a partir do dispositivo de entrada padrão (que na grande maioria das vezes é o teclado).

```
1     // Programa 6: Bemvindo.java
2     // Faz a leitura do nome do usuario e mostra uma mensagem de.
3     // boas vindas.
4
5     import java.util.Scanner;
6
7     public class Bemvindo {
8
9           // o metodo main inicia a execucao do aplicativo Java
10          public static void main(String[] args)
11          {
12                Scanner entrada;
13                entrada = new Scanner(System.in);
14
15                String nome = entrada.nextLine();
16
17                System.out.print("Bem-vindo ");
18                System.out.println(nome);
19
20          } // fim do metodo main
21
22    } // fim da classe Bemvindo
```

Programa 6: Algoritmo 11 implementado em Java.

Na linha 15 é feita a leitura propriamente dita. Note que ela é feita em duas partes. Primeiro, existe a chamada ao método *nextLine* do objeto do tipo *Scanner* dentro da variável *entrada*. Este método espera até que o usuário digite um valor do tipo literal (ou *String* em Java), e após o usuário pressionar a tecla *Enter*, devolve o valor para o programa. A Tabela 5 mostra uma listagem dos métodos de leitura de dados da classe *Scanner*. Existe um para cada tipo de dado. Por fim, na segunda etapa, o valor retornado pelo método *nextLine* é atribuído à variável *nome*, declarada na mesma instrução.

Método	***Descrição***
`nextBoolean`	Lê um valor lógico (*true* ou *false*).
`nextByte`	Lê um número inteiro e retorna um valor do tipo *byte*.
`nextDouble`	Lê um número real (com vírgula ou ponto decimais, de acordo com a configuração de país no computador em que o programa é executado) e retorna um valor do tipo *double*.
`nextFloat`	Similar a nextDouble, mas retorna um valor do tipo *float*.
`nextInt`	Lê um número inteiro e retorna um valor do tipo *int*.
`nextLine`	Lê uma linha de texto e retorna um valor do tipo *String*.
`nextLong`	Lê um número inteiro e retorna um valor do tipo *long*.
`nextShort`	Lê um número inteiro e retorna um valor do tipo *short*.

Tabela 5: Métodos de leitura de dados da classe Scanner.

A segunda forma de obtermos dados a partir de um dispositivo de entrada, é utilizando a classe *JOptionPane*, que já utilizamos para mostrar mensagens em caixas de diálogo. O Programa 7 mostra o Programa 6 modificado para utilizar entrada por caixas de diálogo.

```
1     // Programa 7: BemvindoGUI.java
2     // Faz a leitura do nome do usuario e mostra uma mensagem de
3     // boas vindas.
4
5     import javax.swing.JOptionPane;
6
7     public class BemvindoGUI {
8
9           // o metodo main inicia a execucao do aplicativo Java
10          public static void main(String[] args)
11          {
12                String nome;
13                nome = JOptionPane.showInputDialog(
14                            "Digite seu nome: ");
15
16                System.out.print("Bem-vindo ");
17                System.out.println(nome);
18
19                System.exit(0);
20          } // fim do metodo main
21
22    } // fim da classe BemvindoGUI
```

Programa 7: Programa Bemvindo modificado para utilizar entrada por caixas de diálogo.

As linhas 13 e 14 de nosso programa fazem a leitura de dados propriamente dita. O método *JOptionPane.showInputDialog* exibe a janela mostrada na Figura 21.

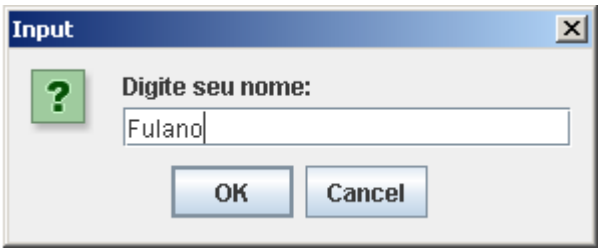


Figura 21: Janela de entrada de dados exibida pelo programa BemvindoGUI.

O argumento para *showInputDialog* indica o que o usuário deve digitar no campo de texto (a caixa com fundo claro). Essa mensagem é chamada de *prompt* porque direciona o usuário para uma ação específica. O usuário digita caracteres no campo de texto e, a seguir, clica no botão *OK* ou pressiona a tecla *Enter* para devolver o texto digitado para o programa. Em nosso programa, o resultado do método será atribuído à variável *nome*.

O método *showInputDialog* de *JOptionPane* sempre retorna um valor do tipo *String*. Se desejamos ler um valor de outro tipo, precisamos *convertê-lo*. Por exemplo, o Programa 8 lê um número inteiro do usuário e o mostra na tela novamente. A novidade neste programa está na linha 15. Declaramos uma variável do tipo *int* chamada *numero*, e desejamos armazenar nela o valor retornado pelo método *JOptionPane.showInputDialog*. Não é possível fazer uma atribuição direta, pois o método devolve um valor do tipo *String*, enquanto a variável é do tipo *int*. Lembre-se que este tipo de atribuição é inválido. Felizmente, a linguagem Java nos possibilita meios de transformarmos um valor do tipo *String* para o tipo *int*. O método *Integer.parseInt* converte seu argumento *String* em um valor do tipo *int*, que agora pode ser armazenado na variável *numero*. A classe *Integer* está definida no pacote *java.lang* (lembre-se que não é necessário importar classes desse pacote). Java oferece outros métodos de conversão, mostrados na Tabela 6.

Método	*Converte de String para*
`Boolean.parseBoolean`	boolean
`Byte.parseByte`	byte
`Double.parseDouble`	double
`Float.parseFloat`	float
`Integer.parseInt`	int
`Long.parseLong`	long
`Short.parseShort`	short

Tabela 6: Métodos de conversão de dados do tipo String para outros tipos.

```
1     // Programa 8: LeNumero.java
2     // Faz a leitura de um numero inteiro do usuario e mostra esse
3     // numero na tela.
4
5     import javax.swing.JOptionPane;
6
7     public class LeNumero {
8
9           // o metodo main inicia a execucao do aplicativo Java
10          public static void main(String[] args)
11          {
12                String numeroDigitado = JOptionPane.showInputDialog(
13                            "Digite um numero inteiro: ");
14
15                int numero = Integer.parseInt(numeroDigitado);
16
17                System.out.print("Voce digitou o numero ");
18                System.out.println(numero);
19
20                System.exit(0);
21          } // fim do metodo main
22
23    } // fim da classe LeNumero
```

Programa 8: Lê um número inteiro do usuário utilizando JOptionPane.showInputDialog e mostra esse número na tela.

4.8. Exercícios

1. Preencha as lacunas em cada uma das seguintes afirmações:

A _____ inicia o corpo do método *main* e a _____ termina o corpo do método.

Cada instrução em Java termina com _____.

_____ inicia um comentário de uma única linha.

A classe _____ contém métodos que exibem caixas de diálogo para entrada e saída de dados.

Os programas em Java iniciam execução pelo método _____.

O método _____ exibe uma mensagem na janela de comando, sem mudar de linha ao final.

2. Marque (V)erdadeiro ou (F)also para cada uma das afirmações abaixo:

( ) Os comentários fazem com que o computador mostre na tela o texto depois das / / quando o programa é executado.

( ) *int* é um dos tipos de dados que representam números inteiros em Java.

( ) Java considera que os nomes *number* e *Number* representam a mesma variável.

( ) O método *Integer.parseInt* converte um inteiro em um *String*.

( ) A palavra-chave *final* serve para declarar constantes em Java.

3. Escreva instruções Java para realizar cada uma das seguintes tarefas:

a) Declarar as variáveis *c*, *thisIsAVariable* e *cti* do tipo float.

b) Exibir uma caixa de diálogo que solicita ao usuário que digite um número inteiro e armazenar o resultado em uma variável chamada *num*.

c) Converter o *String* “123.34” em um número real do tipo *double* e armazenar o valor obtido na variável *medida*.

d) Mostrar a mensagem “Este é um programa em Java.” em uma linha na janela de comando.

e) Mostrar a mensagem “Este é um programa em Java” em duas linhas na janela de comando; a primeira linha deve terminar com a palavra Java. Utilize apenas uma instrução.

4. Quais das seguintes instruções Java contêm variáveis cujos valores são alterados ou substituídos?

a) `p = 7;`

b) `JOptionPane.showMessageDialog(null, "a = 5");`

c) `stringVal = JOptionPane.showInputDialog("Informe um literal: ");`

5. Escreva um programa em Java que exiba os números de 1 a 4 na mesma linha, com cada par de números adjacentes separados por um espaço. Escreva o programa utilizando os seguintes métodos:

a) Uma instrução *System.out*.

b) Duas instruções *System.out*.

6. Transcreva para Java os algoritmos que você escreveu para os exercícios 2, 8 e 9 do capítulo 3.

7. O que a seguinte instrução mostra?

```
System.out.println( "*\n**\n***\n****\n*****" );
```

8. O que a seguinte porção de programa mostra?

```
System.out.print( "*" );
System.out.print( "***" );
System.out.print( "*****" );
System.out.print( "****" );
System.out.println( "**" );
```

9. Utilizando as ferramentas que você aprendeu neste capítulo, escreva um programa em Java que mostre uma tabela com os quadrados e cubos dos números de 0 a 5, como mostrado abaixo:

```
numero  quadrado  cubo
0       0         0
1       1         1
2       4         8
3       9         27
4       16        64
5       25        125
```

5. Introdução às Expressões

O conceito de expressão em termos de programação de computadores é bastante semelhante ao de expressão (ou fórmula) matemática. Nestas últimas, um conjunto de variáveis e constantes numéricas relacionam-se por meio de operadores aritméticos compondo uma fórmula que, uma vez avaliada (calculada), resulta num valor.

Por exemplo, a fórmula de cálculo da área de um triângulo é dada por:

AREA = 0.5 x B x H

Esta fórmula utiliza três variáveis: B e H (base e altura do triângulo, respectivamente), que contém as dimensões do triângulo, e AREA, onde é guardado o valor calculado (resultado da avaliação da expressão). Há, também, um valor constante real (0.5) e o operador de multiplicação (x), que aparece duas vezes na expressão.

O conceito de expressão na programação de computadores assume um aspecto mais amplo: uma expressão em um programa pode ser uma combinação de variáveis e constantes de qualquer tipo, junto com os operadores adequados. Assim como na matemática, uma expressão computacional, depois de avaliada, resulta em um valor.

Iremos utilizar as expressões em algoritmos, e depois em nossos programas, para realizar cálculos e testes. As expressões poderão ser usadas onde um valor de determinado tipo, que poderia ser representado por uma constante ou variável, for esperado.

5.1. Operadores

Operadores são elementos funcionais que atuam sobre operandos e produzem um determinado resultado. Por exemplo, a expressão 3 + 2 relaciona dois operandos (os números 3 e 2) por meio do operador "+" que representa a operação de adição.

De acordo com o número de operandos sobre os quais os operadores atuam, estes podem ser classificados em:

- *binários*, quando atuam sobre dois operandos, como por exemplo os operadores das operações aritméticas básicas (soma, subtração, multiplicação e divisão);
- *unários*, quando atuam sobre um único operando, como por exemplo o sinal de "-" na frente de um número, cuja função é inverter o sinal de seu operando, o próprio número.

A Tabela 7 mostra alguns operadores disponíveis em pseudocódigo, que aprenderemos neste capítulo. Note a semelhança de alguns operadores com a matemática.

Operador	*Símbolo*	*Tipo*	*Para que serve*	*Precedência*
Operadores Aritméticos				
Inversão de sinal	-	Unário	Inverte o sinal de um valor numérico.	7
Manutenção de sinal	+	Unário	Mantém o sinal de um valor numérico.	7
Divisão	/	Binário	Divide dois valores numéricos quaisquer.	6
Multiplicação	*	Binário	Multiplica dois valores numéricos.	6
Resto da divisão	mod	Binário	Devolve o resto da divisão entre dois inteiros.	6
Adição	+	Binário	Soma dois valores numéricos.	5
Subtração	-	Binário	Subtrai dois valores numéricos.	5
Operadores Literais				
Concatenação	+	Binário	Concatena (junta) dois valores literais em um único literal.	5

Tabela 7: Operadores disponíveis em pseudocódigo que estudaremos neste capítulo.

5.2. Tipos de Expressões

As expressões são classificadas de acordo com o tipo de valor resultante de sua avaliação. Em algoritmos e em Java temos três tipos de expressões: aritméticas (que englobam valores reais e inteiros), lógicas e literais. Veremos, neste capítulo, as expressões aritméticas e literais. Veremos as expressões lógicas mais adiante.

5.2.1. Expressões Aritméticas

Expressões aritméticas são aquelas cujo resultado da avaliação é do tipo numérico, ou seja, inteiro ou real. Somente o uso de operadores aritméticos (+, -, *, /, mod) e operandos (variáveis, constantes ou valores) numéricos é permitido em expressões deste tipo.

Os operandos usados em expressões aritméticas podem somente ser do tipo inteiro ou real. Se todos os operandos que aparecem numa expressão são do tipo inteiro, então o valor resultante da avaliação da expressão é também do tipo inteiro. Se ao menos um dos operandos for do tipo real, então o valor resultante do cálculo da expressão será do tipo real.

Nos exemplos seguintes, assumiremos que:

- A e B são variáveis do tipo inteiro;
- X e Y são variáveis do tipo real.

1) A + B * 10 - expressão de resultado inteiro
2) A + B + 10.0 - expressão de resultado real
3) A / B - expressão de resultado inteiro
4) X / Y - expressão de resultado real

Pode parecer estranho o fato de no terceiro exemplo ser apresentada uma expressão onde se dividem dois números inteiros e se obtém como resultado um valor inteiro. Obviamente, o resultado matemático da divisão entre dois números inteiros não é necessariamente inteiro. Por exemplo, se os valores das variáveis A e B fossem, respectivamente, 7 e 2, o resultado matemático esperado seria 3.5, mas em um algoritmo obteríamos o valor 3. Na verdade, a operação representada no terceiro item é a divisão inteira, onde o resto da divisão é desprezado. Para obtermos o resto de uma divisão inteira, usamos o operador *mod*. Por exemplo, 7 mod 2 resulta no valor 1.

Expressões aritméticas são freqüentemente utilizadas em algoritmos. O Algoritmo 13 é um algoritmo para cálculo do quadrado de um número. Note que o número é lido do usuário e armazenado na variável real *num*. Em seguida, existe uma instrução e atribuição para a variável *quad*. Lembre-se: expressões aritméticas podem ser usadas em qualquer lugar em que podemos colocar um valor numérico. Nesta instrução, a expressão será calculada primeiro, e após o valor obtido será atribuído à variável *quad*. Por fim, o valor da variável *quad* é exibido para o usuário.

```
Algoritmo Quadrado1;

var
      num, quad : real;

inicio
      leia num;
      quad ← num * num;
      escreva quad;
fim.
```

Algoritmo 13: Cálculo do quadrado de um número real.

O Algoritmo 14 usa uma expressão para calcular a soma de dois números inteiros:

```
Algoritmo SomaInteiros;

var
      n1, n2, soma : inteiro;

inicio
      leia n1, n2;
      soma ← n1 + n2;
      escreva soma;
fim.
```

Algoritmo 14: Cálculo da soma de dois números inteiros.

5.2.2. Expressões Literais

Expressões literais são aquelas cujo resultado é um valor literal. A única operação disponível é a *concatenação* de literais: toma-se dois literais ou um literal e um valor de outro tipo de dado e acrescenta-se o segundo deles ao final do primeiro, gerando um novo literal. Esta operação é representada pelo símbolo "+".

Por exemplo, a concatenação dos literais "REFRIGERA" e "DOR" é representada por "REFRIGERA" + "DOR" e o resultado é "REFRIGERADOR". Outros exemplos de concatenação:

1.2 + "TESTE" = "1.2TESTE"

"TESTE" + 34 = "TESTE34"

"TESTE" + '!' = "TESTE!"

"TESTE" + .F. = "TESTE.F."

As expressões literais são utilizadas, principalmente, para a montagem de mensagens mais complexas para o usuário, envolvendo tipos diferentes de dados. Por exemplo, vamos modificar o Algoritmo 13, que calcula o quadrado de um número, para que mostre uma pequena mensagem indicando que o número mostrado ao final é o resultado do cálculo do quadrado. Caso o usuário digitasse o número 3, ao invés de mostrarmos somente 9, poderíamos desejar uma informação mais completa como: "O quadrado de 3 é 9". Poderíamos utilizar várias intruções escreva, mas podemos fazer tudo em uma só, utilizando uma expressão literal, como mostrado no Algoritmo 15. A mensagem será montada da esquerda para a direita, concatenando-se primeiro o literal "O quadrado de " com o valor da variável real *num*. O literal resultante será então concatenado com o valor literal " é " e o resultado finalmente concatenado com o valor da variável *quad*.

```
Algoritmo Quadrado2;

var
      num, quad : real;

inicio
      leia num;
      quad ← num * num;
      escreva "O quadrado de " + num + " é " + quad;
fim.
```

Algoritmo 15: Cálculo do quadrado modificado, utilizando uma expressão literal para montagem de uma mensagem mais clara para o usuário.

5.3. Avaliação de Expressões

Expressões que apresentam apenas um único operador podem ser avaliadas diretamente. No entanto, à medida que as mesmas vão se tornando mais complexas, com o aparecimento de mais de um operador numa mesma expressão, é necessária a avaliação passo a passo. Cada subexpressão deve ser avaliada de uma vez, uma após a outra. A seqüência da avaliação é definida pelo formato geral da expressão, considerando-se a prioridade (precedência) de avaliação de seus operadores e a existência ou não de parênteses na mesma.

As seguintes regras são essenciais para a correta avaliação de expressões:

1. Deve-se observar a precedência dos operadores, conforme mostrado na Tabela 7: operadores de maior precedência devem ser avaliados primeiro. Se houver empate em relação à precedência, então a avaliação se faz considerando-se a expressão da esquerda para a direita.
2. Os parênteses usados em expressões se comportam como na matemática, ou seja, têm poder de "roubar" a precedência dos demais operadores, forçando a avaliação da subexpressão em seu interior primeiro.

Exemplos: suponha que X, Y e Z são variáveis reais com os seguintes valores: X = 2.0, Y = 3.0 e Z = 0.5. Considere as seguintes expressões e o resultado de suas avaliações:

1. X * Y – Z = 5.5
2. X * (Y – Z) = 5.0
3. X + Y * Z = 3.5
4. X + (Y * Z) = 3.5
5. (X + Y) * Z = 2.5

Das expressões acima, podemos observar que:

- a aplicação da regra 2 faz com que a expressão 2 tenha um resultado diferente da expressão 1. O mesmo acontece com as expressões 3 e 5;
- o agrupamento de subexpressões usando parênteses contendo operadores de prioridade maior que a dos operadores que permanecem fora dos parênteses, não altera a ordem de avaliação das expressões e, portanto, não modifica seus resultados, como acontece em 3 e 4.

Um último cuidado que devemos tomar: confundir o operador + utilizado para concatenação de literais com o operador + utilizado para adição pode levar a resultados estranhos. Por exemplo, suponha que a variável inteira *y* tem o valor 5. A expressão *"y + 2 =" + y + 2* resulta no literal "y + 2 = 52", e não "y + 2 = 7" porque, o valor de y é concatenado com o literal "y + 2 = ", e então o valor 2 é concatenado com o literal resultante, "y + 2 = 5". A expressão *"y + 2 = " + (y+2)* produz o resultado desejado.

5.4. Expressões Aritméticas e Literais em Java

As expressões em Java funcionam da mesma forma que vimos nos algoritmos. A Tabela 8 mostra o conjunto de operadores aritméticos e literais de Java, em comparação com os algoritmos. Note que a única diferença é o operador *mod*, que em Java é representado pelo símbolo de porcentagem (%).

O Programa 9 contém a implementação do Algoritmo 15, que calcula o quadrado de um número real. Note que as expressões utilizadas têm a mesma forma do algoritmo.

5.5. Interface de um Algoritmo/Programa

O algoritmo de cálculo da soma que vimos (Algoritmo 14) não está perfeito. Em sua forma atual, ao início de sua execução, ele procura ler os valores para as variáveis *n1* e *n2*. Um usuário diferente daquele que criou o programa, a não ser que esteja bem treinado no uso do mesmo, poderá

encontrar dificuldades na interação com o programa. Ele pode confundir a ordem em que os dados devem ser informados, ou simplesmente esquecer o que o programa deseja que digite (pois na tela o que aparece é apenas o cursor piscando, a espera que algo seja digitado). Ao término da execução, o programa escreve como resultado um número que pode não possuir nenhum significado ao usuário se este não souber a finalidade para a qual o algoritmo/programa foi concebido.

Operador	*Símbolo em Algoritmos*	*Símbolo em Java*	*Tipo*	*Precedência*
Operadores Aritméticos				
Inversão de sinal	-	-	Unário	7
Manutenção de sinal	+	+	Unário	7
Divisão	/	/	Binário	6
Multiplicação	*	*	Binário	6
Resto da divisão	mod	%	Binário	6
Adição	+	+	Binário	5
Subtração	-	-	Binário	5
Operadores Literais				
Concatenação	+	+	Binário	5

Tabela 8: Operadores aritméticos e literais da linguagem Java.

```
1     // Programa 9: Quadrado.java
2     // Calcula o quadrado de um numero real.
3
4     import javax.swing.JOptionPane;
5
6     public class Quadrado {
7
8           public static void main(String[] args)
9           {
10                String numeroDigitado = JOptionPane.showInputDialog(
11                            "Digite um numero real: ");
12
13                double num = Double.parseDouble(numeroDigitado);
14
15                double quad = num * num;
16                JOptionPane.showMessageDialog(null,
17                      "O quadrado de " + num + " é " + quad);
18
19                System.exit(0);
20          } // fim do metodo main
21
22    } // fim da classe Quadrado
```

Programa 9: Implementação do algoritmo que calcula o quadrado de um número.

Uma preocupação constante de um bom programador deve ser a de conceber um programa "amigo do usuário". Esta preocupação é traduzida no planejamento de uma *interface* com o usuário (meio pelo qual um programa e o usuário "conversam") bastante amigável. Em termos práticos, isto se resume à aplicação de duas regras básicas:

- toda vez que um programa estiver esperando que o usuário forneça a ele um determinado dado (operação de leitura), ele deve antes enviar uma mensagem dizendo ao usuário o que ele deve digitar, por meio de uma instrução de saída de dados;
- antes de enviar qualquer resultado ao usuário, um programa deve escrever uma mensagem explicando o significado do mesmo.

Uma versão melhorada do Algoritmo 14 é mostrada abaixo (Algoritmo 16). O Programa 10 mostra a implementação deste algoritmo em Java. Um exemplo de execução do programa é mostrado na Figura 22.

```
1     // Programa 10: SomaInteiros.java
2     // Soma dois numeros inteiros.
3
4     import java.util.Scanner;
5
6     public class SomaInteiros {
7
8           public static void main(String[] args)
9           {
10                Scanner entrada;
11                entrada = new Scanner(System.in);
12
13                // Mostra um titulo inicial.
14                System.out.println("--- Cálculo da soma de dois "+
15                      "números ---");
16
17                // Le o primeiro numero
18                System.out.print("Digite o primeiro número: ");
19                int n1 = entrada.nextInt();
20
21                // Le o segundo numero
22                System.out.print("Digite o segundo número: ");
23                int n2 = entrada.nextInt();
24
25                // Calcula a soma dos dois numeros e armazena na
26                // variavel soma.
27                int soma = n1 + n2;
28
29                // Mostra o resultado na tela.
30                System.out.print(n1 + " + " + n2 + " = " + soma);
31
32          } // fim do metodo main
33
34    } // fim da classe SomaInteiros
```

Programa 10: Implementação do algoritmo de soma de inteiros (Algoritmo 16).

```
Algoritmo SomaInteiros2;

var
     n1, n2, soma : inteiro;

inicio
     escreva "--- Cálculo da soma de dois números ---";
     escreva "Digite o primeiro número: ";
     leia n1;
     escreva "Digite o segundo número: ";
     leia n2;
     soma ← n1 + n2;
     escreva n1 + " + " + n2 + " = " + soma;
fim.
```

Algoritmo 16: Algoritmo para cálculo da soma de dois números melhorado.

```
--- Calculo da soma de dois numeros ---
Digite o primeiro numero: 7
Digite o segundo numero: 12
7 + 12 = 19
```

Figura 22: Exemplo de saída do programa de soma de inteiros (Programa 10).

5.6. Exercícios

1. Assinale o tipo do resultado de cada uma das expressões abaixo. Utilize I para inteiro, R para real e L para literal.

( ) 1 + 1	( ) 'A' + "u"	( ) 1 / 2
( ) 1 + " = x"	( ) 1.23 * 100	( ) 0 * 10.5
( ) "2 * " + 10	( ) 4 / 15.0	( ) 5 mod 2

2. Dado que $y = ax^3 + 7$, quais das seguintes expressões são corretas para a equação:

a) a * x * x * x + 7

b) a * x * x * (x + 7)

c) (a * x) * x * (x + 7)

d) (a * x) * x * x + 7

e) a * (x * x * x) + 7

f) a * x * (x * x + 7)

3. Calcule e escreva o resultado de cada expressão abaixo:

a) 7 + 3 * 6 / 2 – 1

b) 2 mod 2 + 2 * 2 – 2 / 2

c) (3 * 9 * (3 + (9 * 3 / (3))))

d) 1 + 2 + " é igual a " + 1 + 1 + 1

4. Complete os algoritmos abaixo com as expressões apropriadas:

```
a) Algoritmo Multiplicacao;
   var
       a, b : real;
       prod : real;
   inicio
        leia a, b;
        prod ← ___________________;
        escreva "O produto dos
números digitados é " + prod;
   fim.
```

```
b) Algoritmo ComprimentoCirculo;
   const
       PI : real = 3.14159;
   var
       raio, compr : real;
   inicio
        leia raio;
        compr ← ___________________;
        escreva "Comprimento = " +
             compr;
   fim.
```

```
c) Algoritmo Horas;
   var
       hora, min : inteiro;
       h_lit     : literal;
       h_min     : inteiro;
   inicio
       escreva "Digite um horário:";
       leia hora, min;

       h_lit ← ____________________;

       h_min ← ____________________;

      escreva "Hora separada por ':' = " + h_lit;
     escreva "Horário em minutos =" + h_min;
   fim.
```

5. Escreva um algoritmo que calcule o cubo de um número real qualquer informado pelo usuário.

6. Escreva um algoritmo para calcular o valor de y na função y = 3x+2.

7. Escreva um algoritmo que leia dois números reais do usuário e mostra a soma, o produto, a diferença e a divisão dos dois números.

8. Escreva um algoritmo para multiplicar duas frações. As frações a serem multiplicadas serão informadas pelo usuário, que deve digitar o numerador e o denominador de cada uma separadamente.

9. Implemente em Java os algoritmos do exercício 4 (com as expressões completadas por você).

10. Implemente, em Java, os algoritmos que você escreveu nos exercícios 5 a 8.

6. Estruturas de Controle I

Nos algoritmos vistos até agora, utilizamos uma estrutura seqüêncial fixa. As mesmas instruções serão executadas todas as vezes em que executarmos o algoritmo, independente dos dados de entrada.

Existem casos em que é necessário executar uma seqüência de instruções diferente no algoritmo, dependendo dos dados de entrada. Por exemplo, dependendo do que um cliente escolher no caixa bancário, uma operação diferente será executada (saque, extrato, etc). Outro fato que pode ocorrer é a necessidade de se executar o mesmo conjunto de instruções várias vezes. Como podemos fazer essas coisas em nossos algoritmos?

Para resolver essa questão existem as chamadas *Estruturas de Controle*. Essas estruturas englobam um ou mais blocos com uma ou mais instruções. Existem, basicamente, dois tipos de estruturas de controle:

- *Estruturas de Decisão*: Nestas estruturas o fluxo de execução é desviado para determinados conjuntos de instruções, dependendo de uma ou mais condições;
- *Estruturas de Repetição:* Servem para repetir um bloco de instruções diversas vezes, sem precisar reescrevê-las.

Neste capítulo daremos uma atenção especial às estruturas de decisão, mais precisamente a estrutura *se...então*. As demais estruturas serão vistas mais adiante em nosso curso.

6.1. Estrutura de Decisão do Tipo Se...Então

As estruturas de decisão são empregadas em situações onde é preciso tomar uma decisão. Dependendo do resultado dessa decisão, um conjunto de instruções será executado em detrimento de outros. Trabalharemos neste capítulo com uma estrutura de decisão em particular: a estrutura *Se...Então*.

A estrutura de decisão *se...então* apresenta a seguinte forma nos algoritmos,

```
se condição então
     conjunto_de_instruções_1
senão
     conjunto_de_instruções_2
fim se;
```

onde:

- *se*, *então*, *senão* e *"fim se"* são palavras-chave que delimitam a estrutura de controle;
- *condição* é uma expressão do tipo lógica que controla para onde o fluxo de execução srá desviado. Caso o resultado dessa expressão seja verdadeiro, o *conjunto_de_instruções_1* será executado. Caso o resultado seja falso, o *conjunto_de_instruções_2* será executado. Veremos mais sobre expressões lógicas na próxima seção;
- *conjunto_de_instruções_1* e *conjunto_de_instruções_2* são seqüências normais de

instruções em pseudocódigo. Esses blocos de instruções podem inclusive conter outras estruturas de controle.

Vejamos um exemplo: como podemos escrever um algoritmo para verificar se uma pessoa é maior de idade ou não, dada a sua idade. A solução é mostrada abaixo (Algoritmo 17):

```
Algoritmo MaiorDeIdade;

var
      idade : inteiro;

inicio
      escreva "Informe a idade a ser verificada: ";
      leia idade;
      se idade >= 18 então
            escreva "Pessoa maior de idade";
      senão
            escreva "Pessoa menor de idade";
      fim se;
      escreva "Fim do programa.";
fim.
```

Algoritmo 17: Algoritmo que lê a idade de uma pessoa e verifica se ela é maior de idade.

Este algoritmo irá mostrar mensagens diferentes, dependendo da idade informada. Caso se informe um valor maior ou igual a 18, a expressão usada na estrutura *se* será verdadeira (veremos mais sobre isso na próxima seção). Logo, o primeiro bloco de instruções será executado e a mensagem "Pessoa maior de idade" será mostrada. O segundo conjunto de instruções (entre *senão* e *fim se*) não será executado. Após, o fluxo de execução retorna para a primeira instrução após o *fim se*. Neste caso, a mensagem "Fim do programa" é mostrada. A saída completa do algoritmo, neste caso, seria como a mostrada abaixo:

```
Informe a idade a ser verificada: 28
Pessoa maior de idade
Fim do programa.
```

Caso se digite uma idade menor do que 18, a estrutura *se* desviará para o segundo bloco de instruções e mostrará a mensagem "Pessoa menor de idade". A saída, neste caso, é mostrada abaixo:

```
Informe a idade a ser verificada: 10
Pessoa menor de idade
Fim do programa.
```

A segunda parte da estrutura *se*, relativa ao *senão*, que é executada quando a condição resulta em um valor .F. (falso), pode ser omitida. Nesses casos, os comandos presentes dentro do *se* só serão executados caso a condição resulte em .V. (verdadeiro). Por exemplo, o algoritmo abaixo (Algoritmo 18) utiliza uma estrutura *se* sem a parte do *senão*:

```
Algoritmo EhZero;

var
      numero : inteiro;

inicio
      escreva "Digite um número inteiro: ";
      leia numero;
      se numero = 0 então
            escreva "Você digitou zero.";
      fim se;
      escreva "O número digitado foi " + numero;
fim.
```

Algoritmo 18: Algoritmo que verifica se um número inteiro digitado é zero.

No algoritmo acima, a mensagem "Você digitou zero" só será mostrada caso o número digitado for zero. A saída do programa, caso o número digitado seja zero é mostrada abaixo:

```
Digite um número inteiro: 0
Você digitou zero.
O número digitado foi 0
```

Caso o usuário digite um número diferente de 0, como -1, a saída será:

```
Digite um número inteiro: -1
O número digitado foi -1
```

Podemos comparar a estrutura *se...então* com uma estrada que apresenta dois caminhos diferentes. Um viajante passando por ela deve decidir por qual deve seguir, a fim de chegar ao destino apropriado. Para isso, pode consultar um mapa, por exemplo. No caso do algoritmo, o mapa consultado é a condição lógica.

6.2. Expressões Lógicas

As expressões lógicas são aquelas cujo resultado da avaliação é um valor lógico (.V. ou .F.). Utilizaremos dois tipos de operadores novos em expressões lógicas: *operadores relacionais* e *operadores lógicos*. A Tabela 9 mostra a listagem completa de operadores disponíveis em pseudocódigo, incluindo os operadores relacionais e lógicos.

Os *operadores relacionais* (=, !=, <, >, <=, >=) são usados quando se deseja efetuar comparações. Comparações só podem ser efetuadas entre variáveis, constantes ou valores do mesmo tipo (com exceção de valores inteiros e reais, que podem ser comparados entre si). O resultado de uma comparação é sempre um valor lógico. Utilizamos operadores relacionais = e >= no Algoritmo 17 e no Algoritmo 18 para montarmos as condições das estruturas *se...então* utilizadas.

Operador	*Símbolo*	*Tipo*	*Para que serve*	*Precedência*
Operadores Aritméticos				
Inversão de sinal	-	Unário	Inverte o sinal de um valor numérico.	7
Manutenção de sinal	+	Unário	Mantém o sinal de um valor numérico.	7
Divisão	/	Binário	Divide dois valores numéricos quaisquer.	6
Multiplicação	*	Binário	Multiplica dois valores numéricos.	6
Resto da divisão	mod	Binário	Devolve o resto da divisão entre dois inteiros.	6
Adição	+	Binário	Soma dois valores numéricos.	5
Subtração	-	Binário	Subtrai dois valores numéricos.	5
Operadores Literais				
Concatenação	+	Binário	Concatena (junta) dois valores literais em um único literal.	5
Operadores Relacionais				
Menor que	<	Binário	Verifica se um valor é menor do que outro.	4
Maior que	>	Binário	Verifica se um valor é maior do que outro.	4
Menor ou igual	<=	Binário	Verifica se um valor é menor ou igual do que outro.	4
Maior ou igual	>=	Binário	Verifica se um valor é maior ou igual do que outro.	4
Igual a	=	Binário	Verifica se um valor é igual a outro.	3
Diferente	!=	Binário	Verifica se um valor é diferente de outro.	3
Operadores Lógicos				
Negação	.não.	Unário	Inverte o valor de um dado ou expressão lógica.	7
Conjunção	.e.	Binário	Conjunção de dois valores lógicos.	2
Disjunção	.ou.	Binário	Disjunção de dois valores lógicos.	1

Tabela 9: Tabela completa de operadores que utilizaremos nos algoritmos.

Outros exemplos: sejam A e B variáveis reais, e R, S e T variáveis literais, com os seguintes valores:

A = 2.5, B = 5.0,

R = “JOSE”, S = “JOAO” e T = “JOAOZINHO”

Abaixo seguem algumas expressões lógicas contendo estas variáveis:

A = B → .F. S = T → .F. S > T → .F.

A = (B / 2) → .V. R != S → .V.

R = S → .F. R > S → .V.

Um resultado interessante a ser observado é o da expressão S > T, ou seja, "JOAO" > "JOAOZINHO", cujo resultado é falso. A primeira pergunta que surge deve ser com relação à maneira como é feita tal comparação. Para tal, deve-se recorrer mais uma vez à tabela de códigos de caracteres (lembra dela?). Tal tabela estabelece para cada caractere um código diferente. De acordo com este código, é possível comparar dois dados de tipo literal ou caractere, comparando os caracteres dos mesmos da esquerda para a direita. Ao comparar os literais "JOSE" e "JOAO", verificamos que seus dois primeiros caracteres ('J' e 'O') são iguais, mas que 'S' é maior do que 'A', segundo qualquer tabela de caracteres. Portanto, "JOSE" é maior do que "JOAO". Estes tipos de comparação são muito úteis na ordenação alfabética de dados literais. Um último aviso: letras maiúsculas e minúsculas são consideradas diferentes. Na tabela ASCII de caracteres, por exemplo, as letras maiúsculas são menores que as minúsculas.

Os *operadores lógicos* (.não., .ou., .e.) são os mais difíceis de entender. Seu funcionamento não é trivial. Para exemplificar seu uso, a Tabela 10 apresenta duas variáveis lógicas A e B com todas as combinações possíveis entre os valores das duas (como só existem dois valores lógicos distintos, essas combinações se reduzem a quatro). As quatro últimas colunas contém os resultados das operações lógicas sobre as combinações possíveis dos valores A e B.

A	***B***	***.não. A***	***.não. B***	***A .ou. B***	***A .e. B***
.F.	.F.	.V.	.V.	.F.	.F.
.F.	.V.	.V.	.F.	.V.	.F.
.V.	.F.	.F.	.V.	.V.	.F.
.V.	.V.	.F.	.F.	.V.	.V.

Tabela 10: Tabela verdade dos operadores lógicos .não., .ou., .e.

Tabelas como a Tabela 10 são chamadas *tabelas-verdade*. Convém salientar as seguintes conclusões que podem ser extraídas por observação da tabela:

- o operador lógico *.não.* sempre inverte o valor de seu operando;
- para que a operação lógica *.ou.* tenha resultado verdadeiro, basta que um de seus operandos seja verdadeiro;
- para que a operação lógica *.e.* tenha resultado verdadeiro é necessário que seus dois operandos tenham valor lógico verdadeiro.

Operadores lógicos aceitam somente operandos lógicos. O Algoritmo 19 exemplifica o uso de operadores lógicos em algoritmos. Este algoritmo verifica se um aluno foi aprovado, a partir de seu conceito (A, B ou C).

Note que existem duas estruturas *se...então*, sendo que a segunda está dentro da primeira. A primeira estrutura *se...então* possui, em sua condição, uma espressão lógica que contém o operador "*.e.*". Essa instrução de decisão verifica se o caractere digitado pelo usuário é inválido, ou seja, se o usuário digitou algo diferente de A, B ou C. Note que a expressão só resultará em verdadeiro nesse caso. Se o caractere digitado for qualquer um dos três, uma das comparações será falsa, causando que toda a condição resulte em falso. Quando desejamos que o algoritmo siga um caminho somente se duas ou mais condições sejam todas verdadeiras, devemos usar o operador *".e."*.

```
Algoritmo EhZero;

var
      conceito : caractere;

inicio
      escreva "Digite seu conceito(A, B ou C): ";
      leia conceito;
      se conceito != 'A' .e. conceito != 'B' .e. conceito != 'C' então
            escreva "Digite apenas A, B ou C.";
      senão
            se conceito = 'A' ou conceito = 'B' então
                  escreva "Aprovado!";
            senão
                  escreva "Reprovado!";
            fim se;
      fim se;
fim.
```

Algoritmo 19: Algoritmo que verifica a aprovação de um aluno, a partir de seu conceito.

A estrutura *se...então* mais interna contém uma expressão lógica que utiliza o operador *".ou."*. Note que o algoritmo só chegará até aqui se a condição do primeiro *"se"* resultar em falso, pois está após o *"senão"*. Este *"se"* mais interno verifica se o aluno tirou A ou B. Caso tenha tirado qualquer um dos dois, está aprovado. Caso contrário, a única opção é o conceito C, sendo ele reprovado. Quando desejamos que o algoritmo siga um determinado caminho se qualquer uma de duas ou mais condições forem verdadeiras, devemos usar o operador *".ou."*.

6.3. Expressões Lógicas em Java

A Tabela 11 mostra a listagem de operadores lógicos e relacionais em Java, comparando os símbolos utilizados na linguagem com os de algoritmos.

Operador	*Símbolo em Algoritmos*	*Símbolo em Java*	*Tipo*	*Precedência*
Operadores Relacionais				
Menor que	<	<	Binário	4
Maior que	>	>	Binário	4
Menor ou igual	<=	<=	Binário	4
Maior ou igual	>=	>=	Binário	4
Igual	=	= =	Binário	3
Diferente	!=	!=	Binário	3
Operadores Lógicos				
Negação	.não.	!	Unário	7
Conjunção	.e.	&&	Binário	2
Disjunção	.ou.	||	Binário	1

Tabela 11: Operadores lógicos e relacionais em Java.

O comportamento e a ordem de precedência dos operadores é exatamente igual aos dos algoritmos. O que difere são os símbolos utilizados para os operadores lógicos e o operador de igualdade. Cuidado! O operador de igualdade em Java é "= =" (sem espaço no meio) e não "=". O sinal "=" é utilizado para a instrução de atribuição. Confundir os dois operadores pode causar erros durante a compilação ou durante a execução do programa (chamados de *erros de sintaxe* e *erros de lógica*, respectivamente).

6.4. A Estrutura Se...Então em Java

A linguagem Java possui uma estrutura de decisão semelhante à estrutura *se...então* dos algoritmos, conhecida como *if...else*. Abaixo é mostrada a forma geral desta estrutura, comparada com sua equivalente nos algoritmos:

```
se condição então
     conjunto_de_instruções_1
senão
     conjunto_de_instruções_2
fim se;
```

```
if ( condição )
     conjunto_de_instruções_1
else
     conjunto_de_instruções_2
```

```
1    // Programa 11: MaiorDeIdade.java
2    // Verifica se uma pessoa eh maior de idade.
3
4    import java.util.Scanner;
5
6    public class MaiorDeIdade {
7
8         public static void main(String[] args)
9         {
10             Scanner entrada;
11             entrada = new Scanner(System.in);
12
13             // Le a idade da pessoa
14             System.out.print("Informe a idade a ser verificada:");
15             int idade = entrada.nextInt();
16
17             if( idade >= 18 )
18                  System.out.println("Pessoa maior de idade");
19             else
20                  System.out.println("Pessoa menor de idade");
21
29             System.out.println("Fim do programa.");
31
32        } // fim do metodo main
33
34   } // fim da classe MaiorDeIdade
```

Programa 11: Implementação do algoritmo que verifica se uma pessoa é maior de idade, a partir de sua idade (Algoritmo 17).

A condição da estrutura *if* deve estar sempre entre parênteses. Esquecê-los é um erro de sintaxe que será acusado durante a compilação do programa. Assim como nos algoritmos, a parte iniciada pela palavra *else* é opcional. O Programa 11 é um exemplo, mostrando uma implementação em Java do algoritmo que verifica se uma pessoa é maior de idade (Algoritmo 17).

As estruturas *if...else*, assim como as estruturas *se...então* nos algoritmos, podem ser *aninhadas*, isto é, podem ser colocadas dentro de outras estruturas *if...else*. As estruturas aninhadas *if...else* podem testar múltiplos casos. Por exemplo, a instrução em pseudocódigo a seguir mostra A para as notas maiores ou iguais a 90, B para as notas no intervalo de 80 a 89, C para as notas no intervalo de 70 a 79, D para as notas no intervalo de 60 a 69 e F para todas as outras notas:

```
se nota >= 90 então
      escreva "A";
senão
      se nota >= 80 então
            escreva "B";
      senão
            se nota >= 70 então
                  escreva "C";
            senão
                  se nota >= 60 então
                        escreva "D";
                  senão
                        escreva "F";
                  fim se;
            fim se;
      fim se;
fim se;
```

Esse pseudocódigo pode ser escrito em Java como:

```
if( nota >= 90 )
      System.out.println("A");
else
      if( nota >= 80 )
            System.out.println("B");
      else
            if( nota >= 70 )
                  System.out.println("C");
            else
                  if( nota >= 60 )
                        System.out.println("D");
                  else
                        System.out.println("F");
```

Se o valor da variável *nota* for maior ou igual a 90, as primeiras quatro condições serão verdadeiras, mas somente a instrução *System.out.println* depois do primeiro teste será executada. Depois de sua execução, a parte *else* da instrução *if...else* externa é "pulada", sem verificar os demais testes (pois estão todos dentro da parte *else* do primeiro *if*) .

A maioria dos programadores Java prefere escrever a estrutura if anterior como:

```
if( nota >= 90 )
      System.out.println("A");
else if( nota >= 80 )
      System.out.println("B");
else if( nota >= 70 )
      System.out.println("C");
else if( nota >= 60 )
      System.out.println("D");
else
      System.out.println("F");
```

Ambas as formas são equivalentes. A última forma é popular porque evita o grande recuo do código para direita. Esse recuo grande freqüentemente deixa pouco espaço em uma linha, forçando quebras de linha e diminuindo a legibilidade do programa.

É importante observar que o compilador Java sempre associa um *else* com o *if* imediatamente anterior a ele, a menos que instruído a fazer de outro modo pela colocação de chaves ({}). Isso é conhecido como *problema do else oscilante*. Por exemplo,

```
if( x > 5 )
      if( y > 5 )
            System.out.println("x e y são > 5");
else
      System.out.println("x é <= 5");
```

parece indicar que, se x for maior do que 5, a estrutura *if* no seu corpo determina se y também é maior do que 5. Se for assim, o literal "x e y são > 5" é enviado para a saída. Caso contrário, parece que, se x não for maior do que 5, a parte *else* da estrutura *if...else* envia para a tela o literal "x é <= 5".

Cuidado! A estrutura aninhada *if* acima não é executada como parece. Na verdade, a estrutura testa se x é maior do que 5. Se for, a execução continua testando se y também é maior do que 5. Se a segunda condição for verdadeira, literal adequado ("x e y são > 5") é exibido. Entretanto, se a segunda condição for falsa, o literal "x é <= 5" é exibido, embora saibamos que x é maior do que 5. Isso ocorre, porque a segunda estrutura *if* é que, na verdade, está mais próxima do *else*. Por isso, o *else* pertence a ela e não à primeira estrutura.

Para forçar a estrutura *if* anterior a ser executada como pretendido, devemos reescrevê-la como segue:

```
if( x > 5 ) {
      if( y > 5 )
            System.out.println("x e y são > 5");
}
else
      System.out.println("x é <= 5");
```

As chaves ({}) indicam ao compilador que a segunda estrutura *if* está no corpo da primeira e que o *else* corresponde à primeira estrutura *if*. As chaves também servem para incluirmos diversas instruções no corpo de um *if* ou *else*. Sem as chaves, apenas uma instrução pode ser inserida. O conjunto de instruções contidas dentro de um par de chaves chama-se *bloco*.

O exemplo abaixo ilustra o uso de chaves para colocarmos duas instruções dentro do corpo de um *else*:

```
if( nota >= 7 )
      System.out.println("Aprovado");
else {
      System.out.println("Reprovado");
      System.out.println("Você deve repetir esta matéria");
}
```

Alguns cuidados que devemos tomar na utilização de estruturas *if* e blocos:

- Colocar um ponto-e-vírgula depois da condição em uma estrutura *if* leva a um erro de lógica quando não há um *else* e a um erro de compilação em estruturas com *else*. O ponto-e-vírgula indica que o *if* termina logo após os parênteses, sendo as instruções internas tratadas como estando após o *if*.
- Variáveis podem ser declaradas dentro de blocos, inclusive dentro de estruturas *if*. Mas cuidado: variáveis declaradas dentro de um *if* só existem ali dentro. Tentar usá-las fora do *if* é um erro, como mostrado abaixo:

```
if( a > 2 ) {
      int x = 10; // A variável x foi declarada dentro do if.
      a = x + 1; // A variável x pode ser usada livremente dentro do if
}

int b = x * 5; // Erro: a variável x não existe aqui
```

6.5. Exercícios

1. Considerando x = 3.0, y = .V. e z = "BRASIL", qual o valor resultante de cada expressão abaixo:

a) x * 8 = 24.0 .ou. z < "ALHO"

b) .não. y .e. x != 7.9

c) (x + 1) / 2 / 5

d) z > "ARGENTINA" .e. y .e. x = x * 3 % 4

2. Escreva um algoritmo que solicita ao usuário que digite dois números inteiros e exibe o maior desses números seguido pelas palavras "é maior". Se os números forem iguais, ele mostra a mensagem "Estes números são iguais".

3. Escreva um algoritmo que lê um inteiro e determina e mostra se ele é ímpar ou par.

4. Escreva um algoritmo que lê dois números inteiros e determina e imprime se o primeiro é um múltiplo do segundo.

5. Quais os erros nos seguintes segmentos de código Java(pode haver mais de um erro em cada segmento):

```
a)
if( genero = 1 )
   System.out.println( "Woman" );
else
   System.out.println( "Man" );
```

```
b)
if( idade >= 65 );
   System.out.println(
    "Idade maior ou igual a 65" );
else
   System.out.println(
     "Idade menor do que 65 )";
```

6. Determine a saída para cada uma das instruções seguintes quando x for 9 e y for 11 e quando x for 11 e y for 9.

```
a) if( x < 10 )
   if( y > 10 )
   System.out.println("*****");
   else
   System.out.println("#####");
   System.out.println("$$$$$");
```

```
a) if( x < 10 ) {
   if( y > 10 )
   System.out.println("*****");
   }
   else {
   System.out.println("#####");
   System.out.println("$$$$$");
   }
```

7. Implemente em Java os algoritmos feitos nos exercícios 2, 3 e 4.

8. Escreva um algoritmo que lê dois horários com horas e minutos e mostre qual deles é maior.

9. Escreva um algoritmo que lê três valores diferentes de zero, determina se eles poderiam ser os lados de um triângulo retângulo e mostra a resposta. Um triângulo retângulo possui um dos lados chamado *hipotenusa* (o maior lado) cujo quadrado é igual a soma dos quadrados dos outros lados.

10. Implemente em Java os algoritmos feitos nos exercícios 8 e 9.

7. Estruturas de Controle II

No início do capítulo anterior, vimos que as estruturas de controle se dividem, basicamente, em dois grupos: *estruturas de decisão* e *estruturas de repetição*. Já estudamos uma estrutura de decisão: a instrução *se...então* (*if...else* em Java). Neste capítulo começaremos o estudo das estruturas de repetição.

São muito comuns as situações em que se deseja repetir um determinado trecho de um algoritmo durante um certo número de vezes. Como exemplo, podemos citar o caso em que se deseja realizar um mesmo processamento para conjuntos de dados diferentes. Exemplo: processamento da folha de pagamentos de uma empresa, em que o mesmo cálculo é efetuado para cada um dos funcionários.

As estruturas de repetição são muitas vezes chamadas de *laços*, ou também, *loops* (laços em inglês).

7.1. Estrutura de Repetição do Tipo Enquanto

A primeira estrutura de repetição que estudaremos é a estrutura *enquanto*. Sua sintaxe é mostrada abaixo:

```
enquanto condição faça
      conjunto_de_instruções
fim enquanto;
```

Esta estrutura funciona da seguinte forma: ao início da instrução *enquanto*, a *condição* é testada. Esta condição deve ser uma expressão lógica qualquer (assim como utilizamos na instrução se...então). Se seu resultado for falso, então o *conjunto_de_instruções* não é executado e a execução do algoritmo prossegue normalmente a partir da instrução seguinte ao *fim enquanto*. Se a condição for verdadeira, o *conjunto_de_instruções* é executado e ao seu término a condição é testada novamente. Assim, o processo será repetido enquanto a condição testada for verdadeira. Quando ela for falsa, o fluxo de execução prosseguirá normalmente pelas instruções após o *fim enquanto*.

Como exemplo inicial de uso da instrução enquanto, vejamos o Algoritmo 20. Este algoritmo calcula a soma dos números de 1 a 20. Quando a execução do algoritmo chega na estrutura enquanto, o valor da variável *contador* é 1 e da variável *soma* é zero. A condição então é testada. A expressão *contador <= 20* resulta em verdadeiro, pois o valor da variável *contador* é 1. Como vimos, se o resultado da condição é verdadeiro, o conjunto de instruções dentro do enquanto será executado. Neste caso, a variável *soma* recebe o resultado da soma de seu conteúdo mais o valor da variável *contador* (0 + 1 = 1). A variável contador recebe o soma de seu conteúdo mais 1 (1 + 1 = 2). Após a execução das instruções do conjunto dentro do enquanto, a condição é novamente testada. Agora *contador* vale 2 e a expressão resulta ainda em verdadeiro. As instruções são novamente executadas. A variável *soma* recebe o resultado de seu conteúdo mais o valor da variável *contador* (1 + 2 = 3). Contador recebe seu conteúdo mais 1 (2 + 1 = 3). A repetição só pára

quando *contador* atingir o valor 21.

```
Algoritmo Soma1a20;
var
      contador, soma: inteiro;
inicio
      soma ← 0;
      contador ← 1;
      enquanto contador <= 20 faça
            soma ← soma + contador;
            contador ← contador + 1;
      fim enquanto;
      escreva "A soma dos números de 1 a 20 é " + soma;
fim.
```

Algoritmo 20: Cálculo da soma dos números de 1 a 20 utilizando uma estrutura enquanto.

A cada repetição do conjunto de instruções dentro da estrutura enquanto, um número diferente entre 1 e 20 é acrescentado à soma (por isso a variável foi inicializada com zero). A cada passo da repetição, o conteúdo da variável contador é aumentado em 1 (na instrução `contador ← contador + 1`). Como veremos na próxima seção, um contador serve para controlarmos quantas vezes o conjunto de instruções deve ser repetido.

Uma vez dentro do laço, a execução somente abandonará o mesmo quando a condição for falsa. O usuário deste tipo de construção deve estar atento à necessidade de que em algum momento a condição deverá ser avaliada como falsa. Caso contrário, o algoritmo permanecerá indefinidamente no interior do laço, o que é conhecido como *laço infinito* ou *loop infinito*.

7.2. Uso de Estruturas de Repetição

Para entendermos o uso de estruturas de repetição em algoritmos, resolveremos alguns problemas, aplicando a cada um uma técnica diferente. Cada uma das subseções a seguir contém a análise de um problema diferente.

7.2.1. Repetição Controlada por Contador

Considere a seguinte definição de problema:

> *Uma turma de dez alunos se submeteu a um teste. As notas (inteiros no intervalo 0 a 100) para esse teste estão disponíveis. Escreva um algoritmo que determine a média da turma no teste.*

A média da turma é igual à soma das notas dividida pelo número de alunos. O algoritmo para resolver esse problema em um computador deve ler cada uma das notas, realizar o cálculo da média e mostrar o resultado.

Podemos utilizar cem variáveis, cada uma para armazenar uma nota, e ler os valores dessas variáveis utilizando uma instrução leia. Essa solução funciona, mas não é muito adequada. Afinal, declarar cem variáveis não é uma tarefa agradável! Utilizaremos uma *repetição controlada por*

contador para ler as notas, uma por vez. Essa técnica utiliza uma variável conhecida como *contador* para controlar o número de vezes que um conjunto de instruções será executado. Neste exemplo, utilizaremos um contador que começa em 1 e terminará em 10, ou seja, a repetição terminará quando o valor do contador exceder 10. A repetição controlada por contador é freqüentemente chamada de *repetição definida*, uma vez que o número de repetições é conhecido antes de o laço começar a ser executado.

O Algoritmo 21 mostra a solução para o problema pronta. Observe as referências no algoritmo a um total e a um contador (representadas pelas variáveis *total* e *cont*, respectivamente). O *total* é uma variável utilizada para acumular a soma de uma série de valores. O *contador* é uma variável utilizada para contar – neste caso, para contar o número de notas lidas. As variáveis utilizadas para armazenar totais devem ser inicializadas com zero antes de serem utilizadas na estrutura de repetição; caso contrário, a soma incluiria o valor anterior armazenado na posição de memória do total (por exemplo, se inicializássemos a variável *total* com o valor 1, ao final do enquanto ela conteria a soma das 10 notas mais 1).

```
Algoritmo Media10Alunos;
var
     cont, total, nota, media: inteiro;
inicio
     total ← 0;
     cont ← 1;
     enquanto cont <= 10 faça
          leia nota;
          total ← total + nota;
          cont ← cont + 1;
     fim enquanto;
     media ← total / 10;
     escreva "A média das notas da turma é " + media;
fim.
```

Algoritmo 21: Cálculo da nota média de uma turma de 10 alunos utilizando repetição controlada por contador.

A condição *cont <= 10* na estrutura enquanto, garante que a repetição irá parar quando a variável *cont*, que representa nosso contador, chegar a um valor maior do que 10. A variável *nota* é utilizada para armazenar cada uma das notas lidas. Veja que após utilizada, cada nota é descartada, pois a variável *nota* é utilizada novamente para a leitura da próxima nota. Como pode a instrução leia ser executada novamente? Note que ela se encontra dentro da estrutura enquanto, e será repetida 10 vezes. A variável *media* é utilizada para armazenar a média inteira das notas dos 10 alunos, ao final da repetição.

7.2.2. Repetição Controlada por Sentinela

Vamos generalizar o problema da média da turma. Considere o seguinte problema:

> *Desenvolver um algoritmo de média da turma que processará um número arbitrário de notas todas vez que o programa for executado.*

No primeiro exemplo de média da turma, o número de notas (10) era conhecido com

antecedência. Nesse exemplo, não se dá nenhuma indicação de quantas notas o usuário irá digitar. O algoritmo deve processar um número arbitrário de notas. Como o algoritmo pode determinar quando parar a leitura de notas? Como ele vai saber quando calcular e mostrar a média da turma?

Uma maneira de resolver esse problema é utilizar um valor especial chamado de *valor de sentinela* (também chamado de *valor de sinalização* ou *flag*) para indicar o final da entrada de dados. O usuário vai digitando até todas as notas válidas terem sido fornecidas. O usuário então digita o valor da sentinela para indicar que a última nota foi fornecida.

Claramente, o valor de sentinela deve ser escolhido de modo que não possa ser confundido com um valor de entrada aceitável. Uma vez que as notas em um teste são normalmente inteiros não negativos, -1 é um valor de sentinela aceitável para esse problema. Portanto, a execução do algoritmo de média da turma poderia processar um conjunto de entradas como 95, 96, 75, 74, 89, e -1. Neste caso, o programa calcularia e mostraria a média da turma para as notas 95, 96, 75, 74 e 89 (-1 é o valor de sentinela, então ele não deve entrar no cálculo da média).

O Algoritmo 22 mostra uma solução em pseudocódigo para o problema. Além de uma variável que acumule a soma total das notas, precisamos também de uma que acumule a quantidade total de alunos da turma (precisamos dessa informação para calcular a média). A variável *alunos* é utilizada para esse fim. Note que ela é inicializada com zero, e a cada repetição do laço enquanto, seu conteúdo aumenta em 1 (instrução `alunos ← alunos + 1`).

Veja que lemos o valor de uma nota antes de iniciar o laço enquanto. Isso é necessário, pois o usuário pode digitar -1 logo na primeira nota (turma sem alunos). Por isso, é necessária a instrução se...então no final do algoritmo, para evitar uma divisão por zero (que não tem valor definido e é inválida em algoritmos). Precisamos, ao final do conjunto de instruções do enquanto, adicionar mais uma instrução leia, para que o algoritmo leia a próxima nota do usuário.

```
Algoritmo Media10Alunos;
var
     alunos, total, nota, media: inteiro;
inicio
     total ← 0;
     alunos ← 0;
     leia nota;
     enquanto nota != -1 faça
          total ← total + nota;
          alunos ← alunos + 1;
          leia nota;
     fim enquanto;
     se alunos > 0 então
          media ← total / alunos;
          escreva "A média das notas da turma é " + media;
     fim se;
fim.
```

Algoritmo 22: Algoritmo que calcula a média de uma turma com um número qualquer de alunos, utilizando repetição controlada por sentinela.

7.3. Instrução *while* em Java

A linguagem Java nos fornece uma estrutura de repetição equivalente à estrutura enquanto

que utilizamos em pseudocódigo: a estrutura *while*. Esta estrutura tem a seguinte sintaxe (compare com a sintaxe da estrutura enquanto em pseudocódigo):

```
while( condição )
{
      conjunto_de_instruções
}
```

Assim como na estrutura *if*, a condição deve sempre ser envolta em parênteses e o uso das chaves não é obrigatório. Mas cuidado, se as chaves não forem utilizadas, o *conjunto_de_instruções* conterá apenas a primeira instrução após o início do *while*.

O Programa 12 apresenta a implementação do Algoritmo 20 em Java, utilizando a estrutura *while*.

```
1     // Programa 12: Soma1a20.java
2     // Soma os numeros de 1 a 20.
3
4     public class Soma1a20 {
5
6           public static void main(String[] args)
7           {
8                 int soma, contador;
9
10                soma = 0;
11                contador = 1;
12
13                while( contador <= 20 ) {
14                      soma = soma + contador;
15                      contador = contador + 1;
16                }
17
18                System.out.println("A soma dos números de 1 a 20 é " +
19                                                soma);
20
21          } // fim do metodo main
22
23    } // fim da classe Soma1a20
```

Programa 12: Implementação em Java do algoritmo que calcula a soma dos números de 1 a 20.

7.4. Atribuição avançada em Java

Java fornece vários operadores de atribuição para abreviar comandos de atribuição. Por exemplo, você pode abreviar a instrução

```
c = c + 3
```

com o *operador de atribuição de adição*, +=, como

```
c += 3
```

O operador += adiciona o valor da expressão à direita do operador ao valor da variável à esquerda do operador, e armazena o resultado na variável à esquerda do operador. Qualquer instrução na forma

variável = variável operador expressão;

onde operador é um dos operadores binários +, -, *, / ou % (ou outros que estudaremos mais tarde), pode ser escrita na forma

variável operador= expressão;

Assim, o comando de atribuição c += 3 adiciona 3 a c. A Tabela 12 mostra os operadores aritméticos de atribuição disponíveis em Java.

Operador de atribuição	***Exemplo***	***Explicação do exemplo***
+=	`c += 7`	`c = c + 7`
-=	`c -= 9`	`c = c - 9`
*=	`c *= 2`	`c = c * 2`
/=	`c /= 10`	`c = c / 10`
%=	`c %= 5`	`c = c % 5`

Tabela 12: Operadores de atribuição aritmética em Java.

7.5. Operadores de Incremento e Decremento em Java

A linguagem Java fornece o operador de incremento unário, ++, e o operador de decremento unário, --, que são resumidos na Tabela 13. O programa pode incrementar (somar 1) o valor de uma variável chamada c por 1 com o operador de incremento ++ em vez dos comandos c = c + 1 ou c += 1.

Os operadores de incremento e decremento podem ser colocados antes ou após o nome de uma variável. Se o operador é colocado antes de uma variável, ele passa a ser chamado de operador de *pré-incremento* ou *pré-decremento*, respectivamente. Se o operador de incremento ou decremento é colocado depois de uma variável, ele passa a ser chamado de operador de *pós-incremento* ou *pós-decremento*, respectivamente.

Operador	*Chamado de*	*Exemplo*	*Explicação do exemplo*
`++`	pré-incremento	`++a`	Incrementa a de 1, depois utiliza o novo valor de a na expressão em que reside.
`++`	pós-incremento	`a++`	Utiliza o novo valor de a na expressão em que reside, depois incrementa a por 1.
`--`	pré-decremento	`--b`	Decrementa b de 1, depois utiliza o novo valor de b na expressão em que reside.
`--`	pós-decremento	`b--`	Utiliza o novo valor de b na expressão em que reside, depois incrementa b por 1.

Tabela 13: Operadores de incremento e decremento em Java.

O Programa 13 demonstra a diferença entre a versão de pré-incremento e a versão de pós-incremento do operador de incremento ++. Pós-incrementar a variável c faz com que ela seja incrementada depois de ser utilizada na chamada do método System.out.println (linha 12). Pré-incrementar a variável c faz com que ela seja incrementada antes de ser utilizada na chamada do método System.out.println (linha 20). A saída do programa é mostrada na Figura 23. O operador de decremento (`--`) funciona de maneira semelhante.

```
1     // Programa 13: Incremento.java
2     // Testa os operadores de pré-incremento e pós-incremento.
3
4     public class Incremento {
5
6           public static void main(String[] args)
7           {
8                 int c;
9
10                c = 5;
11                System.out.println( c );         // mostra 5
12                System.out.println( c++ );       // mostra 5 depois
13                                                 //    incrementa
14                System.out.println( c );         // mostra 6
15
16                System.out.println();            // pula uma linha
17
18                c = 5;
19                System.out.println( c );         // mostra 5
20                System.out.println( ++c );       // pré-incrementa
21                                                 //    depois mostra 6
22                System.out.println( c );         // mostra 6
23
24          } // fim do metodo main
25
26    } // fim da classe Incremento
```

Programa 13: Programa que exemplifica o uso de operadores de pré-incremento e pós-incremento.

```
5
5
6

5
6
6
```

Figura 23: Saída do programa que testa os operadores de pré-incremento e pós-incremento(Programa 13).

Os operadores aritméticos de atribuição e os operadores de incremento e decremento podem ser usados para simplificar instruções de um programa. Por exemplo, as duas instruções de atribuição do Programa 12 nas linhas 14 e 15

```
soma = soma + contador
contador = contador + 1
```

podem ser escritas de maneira mais concisa com operadores de atribuição e pós-incremento como

```
soma += contador;
contador++;
```

É importante aqui comentar que, ao incrementar ou decrementar uma variável em uma instrução isolada, as formas de pré-incremento e de pós-incremento têm o mesmo efeito e as formas de pré-decremento e pós-decremento têm o mesmo efeito. Somente quando uma variável aparece no contexto de uma expressão maior é que pré-incrementar e pós-incrementar a variável têm efeitos diferentes (e igualmente para pré-decrementar e pós-decrementar).

7.6. Exercícios

1. Escreva um algoritmo que calcule a média de um conjunto de números inteiros positivos digitados pelo usuário. A leitura dos números deve terminar quando o usuário digitar o número -2.

2. Crie um algoritmo para exibir os valores da função $F(x) = x^2-5x+6$ para valores de x digitados pelo teclado, até que o usuário digite zero para x.

3. Escreva um algoritmo para calcular o fatorial de um número inteiro digitado pelo usuário. O fatorial de um número é a multiplicação de todos os números inteiros positivos até o número. Por exemplo, o fatorial de 5 é 1x2x3x4x5 = 120.

4. Escreva um algoritmo que verifique quantos alunos, de uma turma de 15, rodaram por freqüência , dadas as freqüências dos alunos como números inteiros de 0 a 100. Um aluno rodou por freqüência se sua freqüência é menor do que 75.

5. Escreva um algoritmo para calcular e mostrar o valor de N^p, sendo N um número real e p um número inteiro positivo ou zero.

6. Identifique e corrija os erros em cada uma das instruções seguintes (pode haver mais de um erro em cada trecho de código):

a)
```
int x = 1, total;
while( x <= 10 ) {
   total += x;
   ++x;
}
```

b)
```
int x = 1, total = 0;
While (x <= 100)
   total += x
   ++x;
```

c)
```
while(y > 0)
   System.out.println( y );
   ++y;
```

7. O que o programa seguinte mostra?

```
public class Mystery {
     public static void main( String[] args ) {
          int y, x = 1, total = 0;
          while ( x <= 10 ) {
               y = x * x;
               System.out.println( y );
               total += y;
               ++x;
          }
          System.out.println("O total é " + total);
     }
}
```

8. Implemente em Java os algoritmos escritos nos exercícios 1 a 5.

8. Estruturas de Controle III

Até aqui já vimos duas estruturas de controle em algoritmos: a estrutura *se...então* e a estrutura *enquanto*. Neste capítulo veremos o restante das estruturas de controle que podemos utilizar em pseudocódigo.

8.1. Estrutura de Decisão do Tipo Escolha

A estrutura do tipo *escolha* oferece a oportunidade de se utilizar mais de duas possibilidades de desvio de fluxo, como a estrutura *se...então* nos possibilita. A partir do valor de uma expressão qualquer, pode-se desviar para um dos N casos especificados. A sintaxe da estrutura *escolha* é mostrada abaixo:

```
escolha expressão
      caso valor1:
            conjunto_de_instruções1
      caso valor2:
            conjunto_de_instruções2
      caso valor3:
            conjunto_de_instruções3
      ...
      senão:
            conjunto_de_instruções_n
fim escolha;
```

onde:

- *escolha* e *fim escolha* são palavras-chave que delimitam a estrutura de controle;
- *expressão* é uma expressão ou variável qualquer cujo valor será utilizado para se desviar para algum dos casos;
- *caso* é uma palavra que designa um caso específico para desvio;
- *valor1*, *valor2*, etc, são os possíveis valores da expressão, no qual existem instruções diferentes a serem executadas. Somente o caso relacionado com o valor exato da expressão será executado, os demais serão ignorados;
- *conjunto_de_instruções1*, *conjunto_de_instruções2*, etc, são os diferentes conjuntos de instruções do algoritmo a serem executados para cada caso;
- *senão* especifica um caso especial, que é executado quando o valor resultante da expressão não está especificado em nenhum caso anterior.

O Algoritmo 23 ilustra a instrução *escolha*. Este algoritmo funciona como uma calculadora

simples, onde algumas operações aritméticas podem ser executadas sobre dois números. Dependendo da operação selecionada pelo usuário (adição, subtração, multiplicação, ou divisão), uma instrução diferente será executada e um valor diferente armazenado na variável *result*. Caso o usuário tenha digitado um caractere diferente dos tratados pelo algoritmo, o valor falso será colocado na variável *op_valido* (pois nenhum caso vai corresponder à expressão e a instrução dentro do senão será executada).

```
Algoritmo Calculadora;
var
     num1, num2, result : real;
     operacao           : caractere;
     op_valido          : lógico;
inicio
     op_valido ← .V.;
     escreva "Digite o primeiro número real: ";
     leia num1;
     escreva "Digite o segundo número real: ";
     leia num2;
     escreva "Digite o símbolo da operação a ser realizada: ";
     leia operacao;

     escolha operacao
          caso '+' : result ← num1 + num2;
          caso '-' : result ← num1 - num2;
          caso '*' : result ← num1 * num2;
          caso '/' : result ← num1 / num2;
          senão:
               op_valido ← .F.;
     fim escolha;

     se op_valido então
          escreva "O resultado da operação foi " + result;
     senão
          escreva "Operação inválida!";
     fim se;
fim.
```

Algoritmo 23: Calculadora simples utilizando a estrutura escolha.

8.2. Estrutura de Repetição do Tipo Para...Faça

A estrutura do tipo *para...faça* é útil quando se conhece previamente o número de vezes que se deseja executar um determinado conjunto de instruções. Logo, este tipo de laço é feito para repetições controladas por um contador, em que contamos o número de vezes em que o corpo de isntruções é executado. A sintaxe usada em pseudocódigos para os laços para...faça é mostrada abaixo:

```
para var de inicio até final incr de inc faça
     conjunto_de_instruções
fim para;
```

A estrutura *para...faça* funciona da seguinte forma: no início da execução do laço o valor *inicio* é atribuído à variável *var*. A seguir, o valor da variável *var* é comparado com o valor *final*. Se *var* for maior que *final*, então o *conjunto_de_instruções* não é executado e a execução do algoritmo prossegue pelo primeiro comando seguinte ao *fim para*. Por outro lado, se o valor de *var* for menor ou igual a *final* então o *conjunto_de_instruções* no interior do laço é executado e, ao final do mesmo, o valor especificado em *inc* é adicionado à variável *var*. Feito isso, retorna-se à comparação entre *var* e *final*, e a execução do *conjunto_de_instruções* será repetida se *var* for menor ou igual a *final*. Quando o laço é finalizado, a execução do algoritmo prossegue pela instrução imediatamente seguinte ao *fim para*.

Logo, a estrutura *para...faça* é equivalente a uma estrutura *enquanto* mostrada abaixo:

```
var ← inicio;
enquanto var <= final faça
      conjunto_de_instruções;
      var ← var + inc;
fim enquanto;
```

O Algoritmo 24 mostra o Algoritmo 20, que calcula a soma dos números de 1 a 20, modificado para utilizar a estrutura *para..faça* em lugar da estrutura *enquanto*. Neste algoritmo, a variável *contador* é utilizada na estrutura *para...faça* para contar os números, começando em 1 até 20. Ao fim de cada repetição do código dentro do *para..faça*, esta variável é incrementada em 1. A cada repetição um novo número é então somado ao conteúdo da variável *soma* e o resultado armazenado novamente na própria variável.

```
Algoritmo Soma1a20;
var
      contador, soma: inteiro;
inicio
      soma ← 0;
      para contador de 1 até 20 incr de 1 faça
            soma ← soma + contador;
      fim para;
      escreva "A soma dos números de 1 a 20 é " + soma;
fim.
```

Algoritmo 24: Algoritmo que calcula a soma dos números de 1 a 20 modificado, utilizando uma estrutura para...faça.

8.3. Estrutura de Repetição do Tipo Repita

Sua sintaxe é mostrada abaixo:

```
repita
      conjunto_de_instruções
até que condição;
```

Seu funcionamento é bastante parecido ao da estrutura *enquanto*. O *conjunto_de_instruções* é executado uma vez. A seguir, a *condição* é testada: se ela for falsa, o *conjunto_de_instruções* é executado novamente e este processo é repetido até que a *condição* seja verdadeira quanto então, a

execução prossegue pelo comando imediatamente seguinte ao final da estrutura.

Esta estrutura difere da estrutura *enquanto* em dois aspectos:

- a repetição termina quando a condição for verdadeira e não falsa, como na estrutura *enquanto*;
- o conjunto de instruções é executado pelo menos uma vez (pois as instruções são executadas antes do teste da condição), ao passo que na estrutura *enquanto* o conjunto de instruções pode não ser executado (caso a condição resulte em falso logo no primeiro teste).

O Algoritmo 25 mostra o exemplo do cálculo da soma dos números de 1 a 20 utilizando uma estrutura *repita*.

```
Algoritmo Soma1a20;
var
     contador, soma: inteiro;
inicio
     soma ← 0;
     contador ← 1;
     repita
          soma ← soma + contador;
          contador ← contador + 1;
     até que contador > 20;
     escreva "A soma dos números de 1 a 20 é " + soma;
fim.
```

Algoritmo 25: Algoritmo para cálculo da soma dos números de 1 a 20 utilizando a estrutura repita.

8.4. Exercícios

1. Escreva um algoritmo que, a partir de um número inteiro de 0 a 10 informado pelo usuário, escreva esse número por extenso. Utilize uma estrutura do tipo escolha para resolver este problema.

2. O Detran deseja realizar uma vistoria geral nos veículos do estado. Para isso, elaborou a seguinte distribuição, a partir do último dígito do número na placa do automóvel:

- Placas que terminam com 0 devem realizar a vistoria em Janeiro / 2009
- Placas que terminam com 1 devem realizar a vistoria em Fevereiro / 2009
- Placas que terminam com 2 devem realizar a vistoria em Março / 2009
- Placas que terminam com 3 devem realizar a vistoria em Abril / 2009
- Placas que terminam com 4 devem realizar a vistoria em Maio / 2009
- Placas que terminam com 5 devem realizar a vistoria em Junho / 2009
- Placas que terminam com 6 devem realizar a vistoria em Setembro deste ano

- Placas que terminam com 7 devem realizar a vistoria em Outubro deste ano
- Placas que terminam com 8 devem realizar a vistoria em Novembro deste ano
- Placas que terminam com 9 devem realizar a vistoria em Dezembro deste ano

Escreva um algoritmo que, a partir do número da placa de um veículo (um inteiro de 4 dígitos), mostre em qual mês ele deve realizar a vistoria. Utilize uma estrutura escolha para isso.

3. Escreva um algoritmo que, utilizando uma estrutura para...faça, calcule e mostre a soma dos números ímpares de 1 a 99.

4. Escreva um algoritmo que calcula o produto dos inteiros ímpares de 1 a 15 utilizando uma estrutura para...faça.

5. Escreva um algoritmo que localiza o menor de vários inteiros positivos. O conjunto de números inteiros informados terminará quando o usuário informar zero. Utilize uma estrutura repita.

6. Escreva um algoritmo para mostrar a quantidade de números ímpares digitados pelo usuário. O processo de digitação de números pelo usuário termina quando o mesmo digitar zero. Utilize uma instrução repita neste algoritmo.

7. A Física Clássica diz que um corpo abandonado de uma determinada altura (H) tem essa altura variando com a seguinte equação

$$\mathbf{H = 0.5 \times G \times T^2}$$

onde G é a aceleração da gravidade, com valor aproximado de 9,806 m/s^2 (ao nível do mar) e T o intervalo de tempo medido em segundos. Considerando esses dados, escreva um algoritmo para calcular e imprimir a altura do corpo nos primeiros 30 segundos de queda. O algoritmo deve mostrar a altura do corpo no instante 0s, 1s, 2s, ... Utilize uma estrutura para...faça neste exercício.

8. Escreva um algoritmo que para gerar os 100 primeiros elementos da *Seqüência de Fibonacci*, em que o próximo elemento é sempre a soma de seus dois anteriores, como segue:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Utilize uma estrutura para...faça.

9. Estruturas de Controle IV

Neste último capítulo sobre estruturas de controle, veremos as últimas três estruturas que estudaremos em Java: a estrutura *for*, a estrutura *do..while* e a estrutura *switch*. Estas três estruturas podem ser utilizadas na implementação de algoritmos que contenham estruturas *para...faça*, *repita* e *escolha*.

9.1. A Estrutura de Decisão *switch*

A estrutura *switch* é o equivalente em Java para a estrutura *escolha* dos algoritmos. O formato geral desta estrutura é:

```
switch( expressão )
{
      case valor1:
            conjunto_de_instruções1;
            break;
      case valor2:
            conjunto_de_instruções2;
            break;
      case valor3:
            conjunto_de_instruções3;
            break;
      ...
      default:
            conjunto_de_instruções_n;
}
```

onde cada *case* corresponde a um *caso* da estrutura escolha e a seção *default* corresponde à seção *senão*. O Programa 14 implementa o Algoritmo 23 utilizando a estrutura *switch*.

A instrução *break*, colocada ao final de cada *case*, faz com que o fluxo de controle do programa prossiga com a primeira instrução depois da estrutura *switch*. Sem *break*, os *case*s em uma estrutura *switch*, vários deles poderiam ser executados de uma só vez. Cada vez que ocorre uma coincidência na estrutura com um *case*, as instruções para todos os *case*s restantes serão executadas. Por exemplo, se não houvéssemos colocado a instrução *break* nos *case*s, e o usuário escolhesse a operação de subtração, as operações de multiplicação e divisão também seriam calculadas.

Uma restrição importante: os valores ao lado de cada *case* podem ser apenas dos tipos inteiros, reais, lógico e caractere. Expressões *switch* que trabalham com valores do tipo *String* não são válidas. Além disso esses valores podem ser representados por valores diretamente digitados desses tipos ou constantes, mas não por variáveis.

```
1     // Programa 14: Calculadora.java
2     // Implementacao de uma calculadora simples.
3
4     import java.util.*;
5
6     public class Calculadora {
7
8           public static void main(String[] args)
9           {
10                Scanner entrada = new Scanner(System.in);
11
12                double num1, num2, result = 0;
13                boolean op_valido;
14                char operacao;
15
16                op_valido = true;
17
18                System.out.print("Digite o primeiro numero real: ");
19                num1 = entrada.nextDouble();
20                System.out.print("Digite o segundo numero real: ");
21                num2 = entrada.nextDouble();
22                System.out.print("Digite o símbolo da operação a"+
23                          " ser realizada: ");
24                operacao = entrada.next().charAt(0); // Le um char
25
26                switch( operacao ) {
27                      case '+' : result = num1 + num2; break;
28                      case '-' : result = num1 - num2; break;
29                      case '*' : result = num1 * num2; break;
30                      case '/' : result = num1 / num2; break;
31                      default: op_valido = false;
32                }
33
34                if(op_valido)
35                      System.out.println("O resultado da operação "+
36                            "foi " + result);
37                else
38                      System.out.println("Operacao invalida!");
39          } // fim do metodo main
40
41    } // fim da classe Calculadora
```

Programa 14: Implementação de uma calculadora simples em Java utilizando a estrutura switch.

9.2. A Estrutura de Repetição *for*

O formato geral da estrutura *for* é

```
for( instruções1 ; condição ; instruções2 )
     instrução
```

onde *instruções1* é um conjunto de uma ou mais instruções, separadas por vírgulas, que são executadas uma única vez antes da execução do laço; *condição* é uma expressão lógica que contém

a condição de continuação da repetição (enquanto a expressão for verdadeira, o laço continua); e *instruções2* contém um conjunto de uma ou mais instruções, separadas por vírgulas, que são executadas após cada repetição das instruções dentro do laço. As três partes do cabeçalho o laço devem ser separadas por um ponto-e-vírgula, como mostrado.

Assim, o laço *for* seria equivalente ao seguinte laço *while*:

```
instruções1;
while( condição ) {
      instrução;
      instruções2;
}
```

A estrutura *for* é freqüentemente utilizada em repetições controladas por contador, na implementação de algoritmos que utilizem a estrutura *para...faça*. Neste caso, *instruções1* contém uma atribuição para a variável contadora do valor inicial e *instruções2* contém uma instrução de incremento ou decremento. O Programa 15 mostra a implementação em Java do Algoritmo 24 utilizando uma estrutura *for*.

```
1     // Programa 15: Soma1a20For.java
2     // Soma os numeros de 1 a 20 utilizando uma estrutura for.
3
4     public class Soma1a20For {
5
6           public static void main(String[] args)
7           {
8                 int contador, soma;
9
10                soma = 0;
11                for( contador = 1; contador <= 20; contador++ )
12                      soma += contador;
13
14                System.out.println( "A soma dos números"
15                      " de 1 a 20 eh " + soma);
16          } // fim do metodo main
17
18    } // fim da classe Soma1a20For
```

Programa 15: Soma dos números de 1 a 20 utilizando a estrutura for.

As linhas 11 e 12 poderiam ser substituídas por uma estrutura *while* equivalente, como mostrado abaixo:

```
contador = 1;
while( contador <= 20 ) {
      soma += contador;
      contador++;
}
```

Assim como na estrutura *while*, quando utilizamos mais de uma instrução dentro de um laço

for, devemos colocá-las entre chaves { }. Os blocos *instruções1* e *instruções2* também podem conter várias instruções, que devem ser separadas por vírgulas. Isto permite ao programador utilizar múltiplas expressões de inicialização e/ou múltiplas expressões de incremento/decremento. Por exemplo, pode haver várias variáveis de controle em uma única estrutura *for* que devem ser inicializadas e incrementadas. As linhas 10, 11 e 12 do Programa 15 poderiam ser substituídas por:

```
for( contador = 1,soma = 0; contador <= 20; contador++,soma += contador);
```

Por razões de clareza, evita-se fazer o que fizemos acima. Coloque apenas expressões e instruções que envolvam os contadores nas seções *instruções1*, *condição* e *instruções2*. As manipulações de outras variáveis devem aparecer antes do laço (se devem ser executadas somente uma vez, como instruções de inicialização) ou no corpo do laço (se devem ser executadas uma vez a cada iteração do laço, como instruções de incremento ou decremento).

A variável contadora pode ser declarada dentro do bloco *instruções1*. Neste caso, a variável ssó pode ser utilizada dentro da estrutura *for* em que foi declarada, como mostrado abaixo:

```
int a = 0, b = 1, c;
for( int cont = 1; cont <= 10; cont++) {
      a += cont;
      b *= cont;
}
c = cont; // ERRO: cont não pode ser usada fora do laço for acima.
```

As três partes do cabeçalho da estrutura *for* são opcionais. Se condição for omitida, Java assume que a condição do laço seja *true*, criando assim um *loop* infinito. Pode-se omitir *instruções1* se o programa inicializa a variável de controle antes do laço, como mostrado abaixo:

```
int a = 0, cont = 1;
for( ; cont <= 10; cont++ )
      a += cont;
```

Pode-se omitir *instruções2* se o programa calcular o incremento com instruções no corpo do laço ou se o laço não exigir um incremento. O exemplo abaixo mostra um laço for em que a última seção do cabeçalho foi omitida:

```
int a = 0;
for( int cont = 1; cont <= 10; ) {
      a += cont;
      cont++;
}
```

9.3. A Estrutura de Repetição *do...while*

A estrutura *do...while* é a implementação em Java da estrutura *repita* de algoritmos, com

uma diferença: a estrutura *do...while* conjunto de instruções dentro dela enquanto a condição especificada for verdadeira, ao contrário do *repita*. O formato geral da estrutura é mostrado abaixo:

```
do {
      conjunto_de_instruções
} while( condição );
```

O Programa 16 implementa o Algoritmo 25 utilizando uma estrutura *do...while*.

```
1     // Programa 16: Soma1a20DoWhile.java
2     // Soma os numeros de 1 a 20 utilizando uma estrutura do/while.
3
4     public class Soma1a20DoWhile {
5
6           public static void main(String[] args)
7           {
8                 int contador, soma;
9
10                soma = 0;
11                contador = 1;
12                do{
13                      soma += contador;
14                      contador++;
15                } while( contador <= 20);
16
17                System.out.println( "A soma dos números"
18                      " de 1 a 20 eh " + soma);
19          } // fim do metodo main
20
21    } // fim da classe Soma1a20DoWhile
```

Programa 16: Calculo da soma dos números de 1 a 20 utilizando uma estrutura do...while.

9.4. As Instruções *break* e *continue*

As instruções *break* e *continue* alteram o fluxo de execução de um programa. A instrução *break*, quando executada em uma estrutura *while*, *for*, *do/while* ou *switch*, causa a saída imediata dessa estrutura. A execução continua a partir da primeira instrução depois da estrutura. O Programa 17 demonstra a instrução *break* em uma estrutura de repetição *for*. A execução desse programa é mostrada na Figura 24.

Quando a estrutura *if* na linha 11 detecta que *cont* é 5, a instrução *break* na linha 12 é executada. Essa instrução termina o laço *for* e o programa prossegue a partir da linha 17. O corpo do laço é executado completamente apenas quatro vezes.

```
1     // Programa 17: ExemploBreak.java
2     // Programa que exemplifica o uso da instrucao break.
3
4     public class ExemploBreak {
5
6           public static void main(String[] args)
7           {
8                 for( int cont = 1; cont <= 10; cont++ ) {
9
10                      // Se a contagem for 5, termina a repeticao
11                      if(cont == 5)
12                            break;
13
14                      System.out.print(cont + " ");
15                }
16
17                System.out.println();
18          } // fim do metodo main
19
20    } // fim da classe ExemploBreak
```

Programa 17: Exemplo de uso da instrução break em um laço for.

```
1 2 3 4
```

Figura 24: Saída do programa de exemplo da estrutura break (Programa 17).

```
1     // Programa 18: ExemploContinue.java
2     // Programa que exemplifica o uso da instrucao continue.
3
4     public class ExemploContinue {
5
6           public static void main(String[] args)
7           {
8                 for( int cont = 1; cont <= 10; cont++ ) {
9
10                      // Se a contagem for 5, termina a repeticao
11                      if(cont == 5)
12                            continue;
13
14                      System.out.print(cont + " ");
15                }
16
17                System.out.println();
18          } // fim do metodo main
19
20    } // fim da classe ExemploContinue
```

Programa 18: Exemplo de uso da instrução continue.

A instrução *continue*, quando executada em uma estrutura *while*, *for* ou *do/while*, pula as instruções restantes do laço e prossegue com a próxima repetição. Nas estruturas *while* e *do/while*, o programa avalia a condição imediatamente depois da instrução *continue* ser executada. Nas estruturas *for*, a expressão de incremento (*instruções2*) é executada e depois o programa avalia a condição.

O Programa 18 mostra um exemplo de uso da estrutura continue. O programa mostra na tela os números de 1 a 10, exceto o número 5. A Figura 25 mostra a saída do programa.

```
1 2 3 4 6 7 8 9 10
```

Figura 25: Saída do programa de exemplo da instrução continue (Programa 18).

9.5. O Operador Condicional '?'

O operador condicional '?' é um operador ternário (trabalha com três operandos) utilizado em algumas situações para substituir uma estrutura *if* simples. A estrutura geral de uso deste operador é mostrada abaixo:

```
condição ? valor1 : valor2
```

onde *condição* é uma expressão lógica e *valor1* e *valor2* são valores, expressões ou variáveis de qualquer tipo. Caso a *condição* seja verdadeira, *valor1* será o resultado; caso contrário, *valor2* será o resultado. O Programa 19 ilustra o uso do operador condicional para verificar se um número é par ou não.

```
1     // Programa 19: ParImparCondicional.java
2     // Programa que exemplifica o uso do operador condicional.
3
4     import java.util.*;
5
6     public class ParImparCondicional {
7
8           public static void main(String[] args)
9           {
10                Scanner entrada = new Scanner(System.in);
11
12                int numero;
13                String resposta;
14
15                System.out.print("Digite um numero inteiro: ");
16                numero = entrada.nextInt();
17
18                resposta = numero % 2 == 0 ? "par" : "impar";
19
20                System.out.println("O numero eh " + resposta);
21          } // fim do metodo main
22
23    } // fim da classe ParImparCondicional
```

Programa 19: Programa que verifica se um número inteiro é par ou ímpar utilizando o operador condicional.

9.6. Exercícios

1. Localize o(s) erro(s) em cada um dos seguintes segmentos de código:

a) O código a seguir deve mostrar se o inteiro value é par ou ímpar:

```
switch (value % 2) {

      case 0:
        System.out.println("Par");

      case 1:
        System.out.println("Impar");
}
```

b) O código seguinte deve gerar como saída os inteiros ímpares de 19 a 1:

```
for(x = 19; x >= 1; x += 2)
      System.out.println( x );
```

c) O código seguinte deve gerar como saída os inteiros pares de 2 a 100 (inclusive):

```
int cont = 2;

do {
      System.out.println( cont );
      cont += 2;
} while( cont < 100);
```

d) Este código gera um loop infinito. Por que?

```
int x = 1;

while(x <= 10);
{
      x++;
}
```

2. O que faz o seguinte programa?

```
public class Printing {
      public static void main(String[] args) {
            for(int i = 1; i <= 10; i++) {
                  for(int j = 1; j <= 5; j++)
                        System.out.print( i % 2 == 1 ? "@" : "#");
                  System.out.println();
            }
      }
}
```

3. Implemente em Java os algoritmos desenvolvidos nos exercícios do capítulo 8. Utilize a estrutura *for* em exercícios que exijam *para...faça*, *do/while* em exercícios com *repita* e *switch* em exercícios com escolha.

10. Procedimentos e Funções

A complexidade dos algoritmos está intimamente ligada à da aplicação a que se destinam. Em geral, problemas complicados exigem algoritmos extensos e de lógica mais complexa para sua solução.

Sempre é possível dividir um problema grande em problemas menores e de solução mais simples. Assim, podemos solucionar cada um destes problemas mais simples e combinar essas soluções de forma que o problema maior seja resolvido.

Por exemplo, um carro é montado utilizando diversas partes (motor, carroceria, pára-brisas, pneus, etc.). Cada parte é fabricada por uma empresa diferente. Durante a fabricação de uma parte, a empresa se foca em resolver o problema específico dela. Um fabricante de pneus se preocupa em fazer pneus mais seguros e resistentes, por exemplo, mas não com a potência do motor. Mas quando se juntam todas as partes, temos um carro completo, que faz mais do que um pneu e um motor.

Assim também podemos fazer com a lógica de um algoritmo. Quando ela é complexa, podemos dividi-la em partes mais simples, e resolver cada parte separadamente. Ao final, podemos juntá-las e obter o algoritmo completo, que resolve o problema maior. Podemos codificar a solução destes problemas simples utilizando os chamados *sub-algoritmos* ou *sub-rotinas*.

Um *sub-algoritmo* é um nome dado a um pequeno algoritmo, que pode ser reutilizado como parte de um algoritmo mais complexo. Por exemplo, um sub-algoritmo que calcule o quadrado de um número pode ser utilizado em um algoritmo que calcule a área de um quadrado, em um algoritmo que calcule o valor de y em uma função de segundo grau x^2+5x-6 dado o valor de x e/ou em um algoritmo que verifique se três números formam lados de um triângulo retângulo.

Em resumo, os sub-algoritmos são importantes na:

- *subdivisão de algoritmos complexos*, facilitando o seu entendimento;
- *estruturação de algoritmos*, facilitando principalmente a detecção de erros e a documentação de sistemas; e
- *modularização de sistemas*, que facilita a manutenção de softwares e a reutilização de sub-algoritmos já implementados.

A idéia de reutilização de software tem sido adotada há muito tempo, por muitos desenvolvedores de sistemas de computador, devido à economia de tempo e trabalho que proporcionam. Seu princípio é o seguinte: um conjunto de algoritmos destinado a solucionar uma série de tarefas bastante corriqueiras é desenvolvido e vai sendo aumentado com o passar do tempo, com o acréscimo de novos algoritmos. A este conjunto dá-se o nome de *biblioteca,* formada por um conjunto de vários sub-algoritmos. Hoje em dia, no desenvolvimento de novos sistemas, procura-se ao máximo basear sua concepção em sub-algoritmos já existentes em uma biblioteca, de modo que a quantidade de software realmente novo que deve ser desenvolvido é minimizada.

Muitas vezes os sub-algoritmos são utilizados para conter certas seqüências de instruções que são repetidas em um algoritmo. Nestes casos, os sub-algoritmos proporcionam uma diminuição no tamanho dos algoritmos.

10.1. Definição de Subalgoritmos em Pseudocódigo

Um algoritmo pode conter diversos sub-algoritmos. Estes devem ser definidos logo após a declaração de variáveis (a não ser que façam parte de alguma biblioteca existente) e antes do início do corpo do algoritmo, como mostrado abaixo:

```
Algoritmo nome_do_algoritmo;


declaração_de_constantes
declaração_de_variáveis


definição_de_subalgoritmos


inicio
      instruções_do_algoritmo
fim.
```

Basicamente, a definição de um sub-algoritmo é formada por duas partes:

- um *cabeçalho*, onde são definidos o *nome* do sub-algoritmo, as *variáveis locais*, os *parâmetros* utilizados e o *tipo* do sub-algoritmo;
- o *corpo* do sub-algoritmo, onde estarão as instruções para a resolução do subproblema.

O *nome* dado a um sub-algoritmo segue as mesmas regras impostas às variáveis e deve ser único. As *variáveis locais* são variáveis definidas e utilizadas dentro do sub-algoritmo. Os *parâmetros* são dados recebidos e/ou enviados ao algoritmo/sub-algoritmo que utiliza a sub-rotina definida.

Basicamente existem dois tipos de sub-algoritmos, diferenciados pelo número de valores retornados ao algoritmo principal:

- as *funções* são sub-algoritmos que retornam um ou mais valores ao algoritmo/sub-algoritmo que o utiliza;
- os *procedimentos* são sub-algoritmos que retornam zero ou mais valores ao algoritmo/sub-algoritmo que o utiliza.

10.2. Funções

O conceito de *função* em algoritmos é fortemente baseado no conceito de função matemática (como seno, cosseno, tangente, etc). Assim, uma função serve para realizar determinado processamento e retornar um dado valor como resultado ao algoritmo/sub-algoritmo que a utiliza.

A sintaxe para definição de funções é mostrada abaixo:

```
função nome_da_função(lista_de_parâmetros): tipo_de_retorno

declaração_de_variáveis_locais

início
     conjunto_de_instruções
fim;
```

O Algoritmo 26 ilustra a utilização de funções em algoritmos. Neste exemplo, a função *Quad* é utilizada para calcular o quadrado de um dado número real.

```
Algoritmo Quadrado;
var
     N : real;

função Quad(num : real) : real
var
     resultado : real;
inicio
     resultado ← num * num;
     retorne resultado;
fim;

inicio
     escreva "Informe um número real para cálculo do quadrado: ";
     leia N;
     escreva "O quadrado do número é " + Quad(N);
fim.
```

Algoritmo 26: Cálculo do quadrado de um número utilizando uma função.

Esta função recebe um parâmetro, chamado *num*, do tipo real. Discutiremos parâmetros em detalhes mais tarde, mas, bascamente, um parâmetro representa um valor qualquer de um tipo específico. No caso de nossa função *Quad*, o parâmetro *num* representa um valor qualquer que será elevado ao quadrado. Quando a função é utilizada no algoritmo, devem ser informados valores para todos os parâmetros definidos.

A função retorna um valor real, que será o resultado do valor do parâmetro *num* elevado ao quadrado. Dentro do corpo da função encontramos uma instrução chamada *retorne*. Essa instrução deve ser utilizada somente dentro de funções e serve para especificarmos o valor de retorno de uma função. Assim, quando executamos uma função, esta retornará o valor especificado pela primeira instrução *retorne* encontrada no fluxo de execução de suas instruções. O valor retornado pode ser especificado como um valor , uma variável, constante ou expressão do mesmo tipo do tipo de retorno especificado.

No fim do corpo do algoritmo, encontramos uma instrução *escreva* onde a função *Quad* é utilizada. Para utilizarmos uma função, basta escrevermos o seu nome seguido por uma listagem de valores separados por vírgulas, entre parênteses. Esses valores serão associados aos parâmetros durante a execução da função.

Quando o fluxo de execução de um algoritmo chega a uma expressão ou comando que

utiliza uma função, o fluxo de execução é temporariamente redirecionado para o início do conjunto de instruções da função. Essas instruções são executadas, até que se chegue a um comando *retorne*. Terminada a execução da função, o fluxo retorna ao ponto onde havia parado antes da execução da função e o valor retornado é utilizada em lugar do nome da função.

Uma observação importante a ser mencionada é que quando uma instrução *retorne* é executada, a execução da função termina, independente se existe ou não instruções dentro da função após o *retorne*. No Algoritmo 27, por exemplo, a instrução *escreva "Mensagem Oculta"* nunca será executada.

```
Algoritmo ExemploRetorne;
var
      numero : inteiro;

função FuncA(N : inteiro) : literal
var
      str : literal;
inicio
      str ← "Teste";
      retorne str + N;
      escreva "Mensagem Oculta";
fim;

inicio
      escreva "Informe um número inteiro: ";
      leia numero;
      escreva FuncA(numero);
fim.
```

Algoritmo 27: Exemplo de instruções não executadas devido ao comando retorne.

10.3. Procedimentos

Um procedimento diferencia-se da função por não se especificar um tipo e retorno. A sintaxe dos procedimentos é mostrada abaixo:

```
procedimento nome_do_procedimento(lista_de_parâmetros)


declaração_de_variáveis_locais


inicio
      conjunto_de_instruções
fim;
```

O Algoritmo 28 é um exemplo de uso de procedimentos em um algoritmo, que utiliza um procedimento para mostrar um histograma de cinco valores na tela. Neste problema, o usuário deve infomar cinco números inteiros positivos. Para cada número informado será desenhada uma linha de asteriscos. Cada linha conterá tantos asteriscos quanto o número informado.

```
Algoritmo Histograma;
var
     n1, n2, n3, n4, n5 : inteiro;

procedimento MostraColuna(N : inteiro);
var
     cont : inteiro;
inicio
     para cont de 1 até N incr de 1 faça
          escreva "*";
     fim para;
fim;

inicio
     escreva "Informe cinco números inteiros positivos: ";
     leia n1, n2, n3, n4, n5;
     MostraColuna(n1);
     MostraColuna(n2);
     MostraColuna(n3);
     MostraColuna(n4);
     MostraColuna(n5);
fim.
```

Algoritmo 28: Desenho de um histograma utilizando um procedimento.

O algoritmo utiliza um procedimento para desenhar cada linha, a partir de um número inteiro. O procedimento é utilizado para mostrar a linha correspondente a cada número informado. Logo, ele é executado cinco vezes, como podemos ver no exemplo. Para executarmos um procedimento basta escrevermos seu nome e a listagem de parâmetros, se houver, entre parênteses. Como os procedimentos não retornam valores, não podem ser usados em expressões, como as funções.

Quando utilizar uma função e quando utilizar um procedimento? Deve-se optarpor uma função quando se tem interesse num valor resultante do processamento. Por exemplo, dado um número, verificar se ele é primo. Neste caso, o mais adequado seria o emprego de uma função cujo retorno é do tipo lógico, pois se deseja uma resposta entre "sim" (.V.) e "não" (.F.). Mas, por outro lado, se o sub-algoritmo não precisa produzir um valor como resultado, o mais adequado é o uso de procedimentos. Um caso seria, por exemplo, mostrar um texto na tela.

Assim como nas funções, quando se executa um procedimento, o fluxo de execução é temporariamente desviado para o corpo do sub-algoritmo. Após o término da sua execução, o fluxo retorna para o primeiro comando após a chamada do procedimento.

10.4. Variáveis Globais e Locais

Variáveis globais são aquelas declaradas no início de um algoritmo. Estas variáveis são visíveis (isto é, podem ser usadas) no corpo do algoritmo e por todos os sub-algoritmos definidos dentro dele.

Variáveis locais são aquelas definidas dentro de um sub-algoritmo e, portanto, somente visíveis (utilizáveis) dentro do mesmo. Outros sub-algoritmos ou mesmo instruções do corpo do

algoritmo não podem utilizá-las.

O Algoritmo 29 ilustra vários casos de uso de variáveis locais e globais. A partir desse exemplo, notamos algumas coisas interessantes:

- as variáveis X e I são globais e visíveis a todos os sub-algoritmos, com exceção da função *Func*, que redefine a variável X localmente;
- as variáveis X e Y, locais à função *Func*, não são visíveis no corpo do algoritmo ou ao procedimento *Proc*. A redefinição local no nome simbólico X como uma variável do tipo literal sobrepõe (somente dentro da função *Func*) a definição global de X como uma variável do tipo real;
- a variável Y dentro do procedimento *Proc*, que é diferente daquela definida dentro da função *Func*, é invisível fora deste procedimento;
- a instrução X ← 3.5 no corpo do algoritmo, bem como as instruções X ← 4 * X e I ← I +1 dentro do procedimento *Proc*, atuam sobre as variáveis globais X e I.

Imagine que um sub-algoritmo é como uma sala fechada. Tudo que é feito ou definido lá dentro só é visto pelas instruções que estão lá dentro. Instruções fora do sub-algoritmo não conseguem enxergar o que existe dentro do sub-algoritmo. As instruções dentro do sub-algoritmo conseguem apenas enxergar as variáveis globais que foram definidas no algoritmo.

```
Algoritmo ExemploVariaveis;
var
      X : real;
      I : inteiro;

função Func() : real
var
      X : literal;
      Y : literal;
inicio
      ...
fim;

procedimento Proc()
var
      Y : literal;
inicio
      ...
      X ← 4 * X;
      I ← I + 1;
      ...
fim;

inicio
      ...
      X ← 3.5;
      ...
fim.
```

Algoritmo 29: Exemplo de uso de variáveis globais e locais.

10.5. Parâmetros

Parâmetros são canais pelos quais se estabelece uma comunicação bidirecional entre um sub-algoritmo e o algoritmo chamador (o algoritmo principal ou outro sub-algoritmo que o utiliza). Dados são passados pelo algoritmo chamador ao sub-algoritmo, ou retornados por este ao primeiro por meio de parâmetros.

Parâmetros formais são os nomes simbólicos instroduzidos no cabeçalho de sub-algoritmos, usados na definição dos parâmetros do mesmo. Dentro de um sub-algoritmo trabalha-se com estes nomes da mesma forma como se trabalha com variáveis locais ou globais. No Algoritmo 30, X e Y são parâmetros formais da função *Media*.

```
Algoritmo MediaComFuncao;
var
      Z : real;

função Media(X, Y : real) : real
inicio
      retorne ( X + Y ) / 2;
fim;

inicio
      Z ← Media( 8, 7 );
      escreva "A média de 8 e 7 é " + Z;
fim.
```

Algoritmo 30: Cálculo da média de dois números utilizando uma função.

Parâmetros formais que tenham o mesmo tipo são listados juntos, separados por vírgula (como mostrado no Algoritmo 30). Caso o sub-algoritmo possua parâmetros com tipos diferentes, estes devem ser separados por ponto-e-vírgula. Por exemplo, se a função *Media* fosse definida com um parâmetro inteiro e um real, o seu cabeçalho seria:

```
função Media( X : inteiro; Y : real) : real
```

Parâmetros reais são aqueles que substituem os parâmetros formais quando da chamada de um sub-algoritmo. Por exemplo, o trecho seguinte, extraído do Algoritmo 30, invoca a função *Media* com os parâmetros reais 8 e 7 substituindo os parâmetros formais X e Y:

```
Z ← Media( 8, 7 );
```

Assim, os parâmetros formais são utilizados somente na definição (formalização) do sub-algoritmo, ao passo que os parâmetros reais substituem-nos a cada invocação do sub-algoritmo. Note que os parâmetros reais podem ser diferentes a cada invocação de um sub-algoritmo.

Um detalhe a ser observado: o parâmetro real a ser utilizado no lugar de um parâmetro formal, durante a invocação de um sub-algoritmo, deve ser de um tipo equivalente. Por exemplo, é um erro passar um dado do tipo literal, como "ABC", no lugar do parâmetro formal X da função *Media*, pois o dado e o parâmetro possuem tipos que não são equivalentes.

10.6. Exercícios

1. Escreva um sub-algoritmo para determinar se um dado número inteiro é ou não ímpar.

2. Escreva um sub-algoritmo para determinar se um número inteiro é ou não positivo.

3. A partir do algoritmo mostrado:

```
Algoritmo Teste;
var
     x, y : inteiro;

função mult(a, b: inteiro) : inteiro
var
     i, result : inteiro;
inicio
     result ← 0;
     para i de 1 até b incr de 1 faça
          result ← result + a;
     fim para;
     retorne result;
fim;

função soma(n1, n2: inteiro) : inteiro
inicio
     retorne n1 + n2;
fim;

procedimento entrada()
inicio
     escreva "--- CALCULOS COM NUMEROS INTEIROS ---";
     escreva "Digite dois números inteiros positivos: ";
fim;

inicio
     entrada();
```

```
        leia x, y;
        escreva "A soma dos número é " + soma(x, y);
        escreva "O produto dos números é " + mult(x, y);
fim.
```

Marque V ou F para as afirmações abaixo:

( ) As variáveis *i* e *result* são visíveis e poderiam ser utilizadas no procedimento *entrada*.

( ) A variável *x* é uma variável local da função *mult*.

( ) Para calcular 5 * 6, a função mult poderia ser utilizada da seguinte forma: *mult(6, 5)*

( ) Os números 0.5 e 6 poderiam ser utilizados como parâmetros reais para a função *soma*.

( ) A variável y é uma variável global.

4. Escreva um procedimento que mostre uma linha com N caracteres. A linha deve ser formada por asteriscos e o sinal de porcentagem (%). A cada três asteriscos deve ser mostrado um sinal de porcentagem. Por exemplo, para uma linha com 10 caracteres, deve ser mostrado: ***%***%**.

5. Escreva um algoritmo que use o procedimento anterior para mostrar linhas com N variando de 1 a 20.

6. Escreva um sub-algoritmo que devolve o valor absoluto de um número real. O valor absoluto de um número é chamado em matemática de módulo, ou seja, o valor do número sem sinal. Por exemplo, o valor absoluto de -3 é 3 e de +5 é 5.

11. Introdução aos Métodos

11.1. Introdução

No capítulo anterior, estudamos como dividir algoritmos em partes menores chamadas sub-algoritmos. Os sub-algoritmos, por sua vez, podem ser de dois tipos: procedimentos ou funções.

Neste capítulo, estudaremos como implementarmos procedimentos e funções em nossos programas em Java, através do uso de um recurso chamado *método*. Um método em Java é um conjunto de instruções que, assim como os procedimentos e funções, pode receber dados externos (os parâmetros) e devolver resultados (como as funções).

Trabalharemos neste capítulo com um tipo especial de métodos em Java, chamado de *métodos estáticos*. Estes métodos são utilizados diretamente a partir das classes em que foram definidos e não necessitam de instâncias para serem executados. Faremos uso de outro tipo de método mais tarde, quando estudarmos uma introdução à programação orientada a objetos.

11.2. Alguns Métodos da API Java

Como vimos no capítulo anterior, é comum aos programadores fazerem uso de bibliotecas de procedimentos e funções já prontos em seus programas. A API Java oferece alguns métodos prontos, que executam tarefas úteis em nossos programas. Todos esses métodos encontram-se organizados em classes, que por sua vez estão organizadas em pacotes.

O primeiro conjunto de métodos que estudaremos é o presente na classe *Math*, presente no pacote *java.lang* (aquele que é importado automaticamente em nossos programas). Os métodos da classe *Math* permitem realizar cálculos matemáticos comuns. Os principais métodos da classe Math são mostrados na Tabela 14. Note que esses métodos se comportam como funções, ou seja, realizam o cálculo e o valor resultante é devolvido.

Um método estático em Java é invocado escrevendo-se o nome da classe, seu nome e depois, entre parênteses a listagem de parâmetros reais. Por exemplo, o programador que deseja calcular a raiz quadrada de 64 poderia escrever

```
Math.sqrt( 64 )
```

O método *sqrt* é um método que se comporta como função. Quando essa instrução é executada, ela invoca o método estático *sqrt* da classe *Math*. O número 64 é o parâmetro real desta chamada do método *sqrt*. O resultado da linha mostrada seria *8.0*. Para escrever esse valor em uma linha do terminal de comando basta escrever a instrução

```
System.out.println( Math.sqrt( 64 ) );
```

Os parâmetros reais de um método podem ser valores, constantes, variáveis ou expressões. Por exemplo, a instrução

```
System.out.println( Math.sqrt( 2 * 6 + 4 ) );
```

calcula e mostra a raiz quadrada de 16, que é 4.

Método	Descrição	Exemplo
`abs( x )`	Valor absoluto de x (x pode ser do tipo *double*, *float*, *int* ou *long*)	`abs(5.1) é 5.1` `abs(-3) é 3`
`ceil( x )`	Arredonda o valor real x para o menor inteiro não menor que x	`ceil(9.2) é 10.0` `ceil(5.0) é 5.0`
`cos( x )`	Cosseno trigonométrico de x (x deve ser informado em radianos)	`cos(0.0) é 1.0`
`exp( x )`	Calcula e^x	`exp(1.0) é 2.71828`
`floor( x )`	Arredonda x para o maior inteiro não maior que x	`floor(9.2) é 9.0` `floor(5.0) é 5.0`
`log( x )`	Logaritmo natural de x (base *e*)	`log(2.71828) é 1.0`
`max( x, y )`	Maior valor entre x e y (x e y podem ser do tipo *double*, *float*, *int* e *long*, sendo que os dois valores devem ser do mesmo tipo)	`max(2.3, 12.7) é 12.7`
`min( x, y )`	Menor valor entre x e y (x e y podem ser do tipo *double*, *float*, *int* e *long*, sendo que os dois valores devem ser do mesmo tipo)	`min(2.3, 12.7) é 2.3`
`pow( x, y )`	Calcula x levado à potência y (x^y)	`pow(2.0,3.0) é 8.0` `pow(9.0,0.5) é 3.0`
`sin( x, y )`	Seno trigonométrico de x (x em radianos)	`sin(0.0) é 0.0`
`sqrt( x )`	Raiz quadrada de x	`sqrt(900.0) é 30.0`
`tan( x )`	Tangente trigonométrica de x (x em radianos)	`tan(0.0) é 0.0`
`random()`	Gera um número real pseudo-aleatório entre 0.0 e 1.0	

Tabela 14: Principais métodos da classe Math.

Vamos agora dar uma olhada mais detalhada em um dos métodos da classe *Math*: o método *random*. Este método *random* gera um valor *double* de 0.0 até, mas não incluindo, 1.0. O método escolhe esse valor como se fosse um sorteio ou um jogo de dados, ou seja, a cada chamada do método um valor *double* diferente, maior ou igual a 0.0 e menor do que 1.0, é retornado.

Observe que os valores devolvidos por *random* são na verdade *números pseudo-aleatórios*, uma seqüência de valores produzida por um cálculo matemático complexo. Este cálculo usa a hora do dia atual para semear o gerador de números aleatórios, de modo que cada execução de um programa dá origem a uma seqüência diferente de valores.

O intervalo de valores produzido diretamente pelo método *random* muitas vezes é diferente do intervalo de valores necessário em um programa Java em particular. Por exemplo, o programa que simula o lançamento de uma moeda talvez exija somente os valores 0 para "cara" e 1 para "coroa". O programa que simula o lançamento de um dado de seis faces exige inteiros aleatórios no intervalo de 1 a 6. Para gerar valores em faixas diferentes, precisamos trabalhar com o valor devolvido pelo método *random*, através de uma expressão aritmética na forma:

```
(int) (valor_inicial + Math.random() * número_de_valores )
```

onde *valor_inicial* é o primeiro valor do intervalo e *número_de_valore*s é o número de valores do intervalo. (int) à frente da expressão faz com que o resultado dela seja convertido para um valor inteiro. Sendo assim, para gerarmos um valor inteiro no intervalo de 1 a 6 basta escrevermos a expressão:

```
(int) (1 + Math.random() * 6 )
```

O Programa 20 usa *Math.random* para simular 5 lançamentos de um dado de 6 faces.

```
1     // Programa 20: SimulaDado.java
2     // Programa que usa Math.random para simular 5 lancamentos de
3     // um dado.
4
5     public class SimulaDado {
6
7           public static void main(String[] args)
8           {
9                 for( int cont = 1; cont <= 5; cont++ ) {
10
11                      int dado = (int)(1 + Math.random() * 6);
12
13                      System.out.println(dado);
14                }
15
16          } // fim do metodo main
17
18    } // fim da classe SimulaDado
```

Programa 20: Simulação do lançamento de um dado 5 vezes utilizando Math.random.

A classe Math também define duas constantes matemáticas comumente utilizadas: *Math.PI* e *Math.E* (respectivamente os números matemáticos π e *e*). Essas duas constantes podem ser usadas diretamente em qualquer lugar onde esses números sejam necessários.

Em capítulos anteriores já utilizamos alguns métodos da API Java. Por exemplo, os métodos de conversão *Double.parseDouble* e *Integer.parseInt*, que fazem parte, respectivamente, das classes *Double* e *Integer*. Outros métodos que utilizamos foram os métodos *showInputDialog* e *showMessageDialog*, da classe *JOptionPane*. Aprenderemos ainda outros métodos da API Java ao longo do curso.

11.3. Criando Métodos

Todos os programas que escrevemos até agora utilizavam uma abordagem chamada *monolítica*. Nesta abordagem, todo o código do programa é escrito em um único conjunto de instruções.

A partir de agora, escreveremos programas que utilizam a abordagem *modular*. Iremos escrever programas que dividem seu código em diversos procedimentos e funções, utilizando o conceito de métodos.

Assim como nos algoritmos, para criarmos um método em Java precisamos defini-lo dentro de uma classe. Como todos os programas em Java são classes, podemos definir métodos dentro das

classes de nossos programas. No próximo capítulo, veremos como definir métodos em outras classes, fora do programa.

O formato geral de uma definição de método estático em Java que utilizaremos neste capítulo é

```
public static tipo_de_retorno nome_do_método ( parâmetros )
{
      instruções
}
```

O termo *public* é uma palavra reservada que estudaremos no próximo capítulo. A palavra *static* é usada para especificar que o método definido é estático. Por enquanto, em todos os métodos que escrevermos, utilizaremos essa palavra. O restante da definição do método é semelhante à definição de uma função na forma

```
função nome_do_método ( parâmetros ) : tipo_de_retorno
início
      instruções
fim;
```

Como todas as definições de métodos exigem um valor de retorno, todos os métodos em Java se comportam como funções. Mas sendo assim, como implementar procedimentos em Java? Para contornar esse problema, existe um tipo especial em Java chamado *void*. O tipo *void*, quando usado como tipo de retorno de um método, especifica que o método não irá retornar valores. Logo, se desejamos implementar um procedimento em Java, basta criarmos um método cujo tipo de retorno é *void*.

A lista de parâmetros na definição de métodos em Java é uma lista separada por vírgulas, na qual o método declara o nome e o tipo de cada parâmetro. Diferentemente dos algoritmos, para cada parâmetro deve ser especificado seu tipo, mesmo que existam vários parâmetros consecutivos com o mesmo tipo. Por exemplo, um método chamado *media* que calcula a média de dois números reais, poderia ser definido da seguinte forma:

```
public static double media ( double a, double b )
{
      ...
}
```

onde *a* e *b* são os nomes dos parâmetros. A declaração

```
public static double media ( double a, b )
{
      ...
}
```

produziria um erro de sintaxe no programa, pois para cada parâmetro deve ser especificado um tipo.

Assim como nos algoritmos, cada parâmetro real passado a um método deve ser compatível com o tipo do parâmetro correspondente na definição do método. Por exemplo, um parâmetro do

tipo *double* pode receber valores 7.35, 22 ou -0.03546, mas não "teste" (porque um valor do tipo *String* não pode ser atribuído a uma variável *double*). Se um método não recebe nenhum valor, a lista de parâmetros será vazia (isto é, após o nome do método há um conjunto vazio de parênteses).

O Programa 21 mostra uma implementação para o Algoritmo 26. Neste exemplo, é definido um método chamado *quad*, que recebe um parâmetro do tipo *double* e retorna o quadrado dele. Veja que o comando equivalente a *retorne* em Java é *return*. O funcionamento do comando *return* é idêntico ao de *retorne*. Na linha 11 é declarada uma variável local ao método *quad*, chamada *resultado*. Assim como no programa, dentro de métodos podemos definir variáveis locais em qualquer lugar dentro do conjunto de instruções do método. Assim como em pseudocódigo, as variáveis locais são visíveis apenas no método onde foram declaradas.

```
1     // Programa 21: Quadrado2.java
2     // Programa que faz uso de um metodo para calcular o quadrado.
3     // de um numero real.
4
5     import java.util.*;
6
7     public class Quadrado2 {
8
9           public static double quad( double num ) {
10
11                double resultado;
12                resultado = num * num;
13                return resultado;
14          } // fim do metodo quad
15
16          public static void main(String[] args)
17          {
18                Scanner entrada = new Scanner(System.in);
19
20                double N;
21
22                System.out.print("Informe um numero real para "+
23                      "calculo do quadrado: ");
24                N = entrada.nextDouble();
25
26                System.out.println("O quadrado do número é "+quad(N));
27
28          } // fim do metodo main
29
30    } // fim da classe Quadrado2
```

Programa 21: Cálculo do quadrado de um número utilizando um método (implementação do Algoritmo 26).

Olhando para a definição de *main*, podemos perceber que ele também é um método estático que recebe um parâmetro e não retorna valor (pois o tipo de retorno é *void*). Vínhamos definindo métodos desde o capítulo 4 e não sabíamos! Todo programa Java é uma classe que define um método chamado *main*. Podemos definir classes sem *main*, como faremos no próximo capítulo, mas essas classes não podem ser executadas como programas.

Como *main* é um método, toda variável declarada dentro dele é uma variável local ao

método. Ou seja, não podemos utilizar variáveis declaradas dentro de *main* em outros métodos da classe. Por exemplo, utilizar a variável *N* declarada no método *main* do Programa 21 seria um erro, pois ela é visível somente dentro do método *main*. Em outras palavras, variáveis declaradas dentro do método *main* não são globais.

Note que na linha 26 não escrevemos o nome da classe antes do nome do método, quando o invocamos. Quando um método é chamado dentro da própria classe em que é definido, isso não é necessário.

11.4. Exercícios

1. Escreva o cabeçalho do método (somente o início da definição do método, sem as instruções) para cada um dos seguintes métodos:

a) O método *hipotenusa*, que recebe dois parâmetros do tipo *double* chamados *cat1* e *cat2* e retorna um valor do tipo *double* como resultado.

b) O método *menor*, que recebe três parâmetros inteiros *x*, *y* e *z* e retorna um valor inteiro como resultado.

c) O método *instrucoes*, que não recebe parâmetros e não retorna valor.

d) O método *intToFloat*, que recebe um parâmetro do tipo *int*, chamado *numero*, e retorna um resultado do tipo *float*.

2. Localize o erro em cada um dos seguintes segmentos de programa e explique como o erro pode ser corrigido:

```
a) public static int sum( int x, int y ) {
            int result;
            result = x + y;
   }

b) public static void f( float a ); {
            float a;
            System.out.println( a );
   }

c) public static void product() {
            int a = 6, b = 5, c = 4, result;
            result = a * b * c;
            System.out.println( "O resultado eh " + result );
            return result;
   }
```

3. Escreva um programa em Java que calcule o volume de uma esfera. O programa deve definir e utilizar um método chamado *volumeEsfera*. Este método deve receber o raio de uma esfera como parâmetro (um número real) e retornar o volume da mesma (um número real). O volume de uma esfera pode ser calculado pela fórmula:

$$\text{Volume} = \frac{4}{3} * \pi * \text{Raio}^3$$

Obs.: Utilize a constante *Math.PI* para o cálculo do volume.

4. Qual é o valor de x depois que cada uma das seguintes instruções é executada?

```
a) x = Math.abs( 7.5 );
b) x = Math.floor( 7.5 );
c) x = Math.abs( 0.0 );
d) x = Math.ceil( 2.7 );
e) x = Math.abs( -6.4 );
f) x = Math.ceil( -5.2 );
g) x = Math.ceil( Math.abs( -8 + Math.floor( -5.5 ) ) );
```

5. Uma aplicação do método *Math.floor* é arredondar um valor para o inteiro mais próximo. A instrução

```
y = Math.floor( x + 0.5 );
```

arredondará o número x para o inteiro mais próximo e atribuirá o resultado a y. Escreva um programa em Java que lê números reais e utiliza a instrução precedente para arredondar cada um desses números para o inteiro mais próximo. Para cada número processado, exiba ambos os números, o original e o arredondado. O usuário informará que terminou de digitar os números, quando entrar com um valor negativo.

6. Escreva instruções que atribuem inteiros aleatórios à variável n nos seguintes intervalos:

a) 1 – 2
b) 1 – 100
c) 0 – 9
d) 1000 – 1112
e) -1 – 1
f) -3 – 11

7. Escreva um método chamado *multiplo* que determina, para dois números inteiros passados como parâmetros, se o segundo número é múltiplo do primeiro. O método deve devolver *true* em caso afirmativo e *false* caso contrário. Incorpore esse método a um programa em Java que lê 5 pares de números inteiros e diz se o segundo é múltiplo do primeiro, utilizando o método.

8. Escreva um método *ehPar* que determina se um número inteiro é par. O método deve receber um parâmetro inteiro e retornar *true* se o inteiro for par e *false* caso contrário. Incorpore esse método

em um programa Java que mostre os números pares entre -50 e 50.

9. Escreva um método *potencia (base, expoente)*, onde a base é um valor real e o expoente um valor inteiro. Não utilize nenhum método da biblioteca matemática de Java. Incorpore esse método a um programa em Java que lê números inteiros do usuário para a base e o expoente e realiza o cálculo com o método especificado. O usuário pode digitar diversos pares de base e expoente. A entrada de dados termina quando for digitados uma base e um expoente iguais a zero. Note que o método deve lidar com o caso de um expoente negativo.

10. Escreva um método chamado *menorDe3* que retorna o menor de três números reais. Utilize o método *Math.min* para implementar *menorDe3*. Incorpore o método a um programa Java que lê três valores do usuário e determina o menor valor.

11. Implemente os seguintes métodos que retornam valores inteiros:

a) O método *celsius* retorna o equivalente em Celsius de uma temperatura em Farenheit com o cálculo:

```
C = 5 / 9 * ( F - 32 )
```

b) O método farenheit retorna o equivalente em Farenheit de uma temperatura em Celsius.

```
F = 9 / 5 * C + 32
```

c) Utilize esses métodos para escrever um programa em Java que permita ao usuário digitar uma temperatura em Farenheit e exiba o equivalente em Celsius ou digite uma temperatura em Celsius e exiba o equivalente em Farenheit.

12. Uso Avançado de Métodos

12.1. Recursão

Os programas que construímos até agora fazem uso de métodos de uma forma disciplinada e hierárquica. Mas uma pergunta pode surgir: é possível chamar um método dentro dele mesmo? A resposta a esta pergunta é sim. E mais: para alguns problemas, é útil fazer os métodos chamarem si mesmos. A um método que chama a si mesmo dá-se o nome de *método recursivo*. Um método recursivo chama a si próprio diretamente ou indiretamente, através de outro método.

Os problemas a resolvidos recursivamente apresentam alguns aspectos em comum. Quando um método recursivo é chamado para resolver um problema, ele o compara com dois *casos*: um caso mais complexo e outro chamado *básico.* Quando o método recursivo é chamado para resolver um caso básico, o método simplesmente devolve um resultado. Mas se o método é chamado a resolver um caso mais complexo, ele o divide em duas partes: uma parte mais simples, que o método sabe resolver, e outra que o método não sabe como resolver. Para tornar a recursão possível, a segunda parte deve ser uma versão menor ou mais simples do problema original. Como essa segunda parte é semelhante ao problema original, o método chama uma nova cópia de si mesmo para ir trabalhar no problema menor; este tipo de procedimento, conhecido como *chamada recursiva*, também é conhecido como *etapa da recursão*. O resultado do método original será uma combinação da solução da parte mais simples, que o método sabia resolver, com a solução da chamada recursiva.

Vamos analisar a recursão na prática utilizando um problema matemático: o cálculo do *fatorial* de um número inteiro não-negativo. O fatorial de um número *n* não-negativo, escrito *n!* é o produto

$$n * (n – 1) * (n – 2) * ... * 1$$

com 1! igual a 1 e 0! igual a 1. Por exemplo, 5! é o produto de 5 * 4 * 3 * 2 * 1, que é igual a 120.

Chega-se a uma definição recursiva do método fatorial observando-se o seguinte relacionamento:

$$n! = n * (n – 1)!$$

Por exemplo, 5! é claramente igual a 5 * 4!, como mostrado a seguir:

5! = 5 * 4 * 3 * 2 * 1

5! = 5 * (4 * 3 * 2 * 1)

5! = 5 * 4!

O Programa 22 implementa recursivamente o cálculo do fatorial, como descrito acima. Vamos nos concentrar no que é novo, ou seja, na implementação do método recursivo *fatorial* (note que o restante do programa não apresenta nenhuma estrutura desconhecida). As linhas 11 e 12 do método apresentam o caso básico, ou seja, quando o valor de *n* for 1 ou 0, o método retornará 1. Nos demais casos, implementados na linha 14, o método retornará o resultado da multiplicação de *n* pelo resultado do fatorial de *n – 1*, implementando assim a etapa de recursão do problema.

```
1     // Programa 22: FatorialRecursivo.java
2     // Programa que faz uso de um metodo recursivo para calcular o
3     // fatorial de um numero.
4
5     import java.util.*;
6
7     public class FatorialRecursivo {
8
9           public static long fatorial( int n ) {
10
11                if(n <= 1)
12                      return 1;
13                else
14                      return n * fatorial(n - 1);
15
16          } // fim do metodo fatorial
17
18          public static void main(String[] args)
19          {
20                Scanner entrada = new Scanner(System.in);
21
22                int num;
23
24                System.out.print("Informe um numero inteiro para "+
24                      "calculo do fatorial: ");
25                num = entrada.nextInt();
26
27                System.out.println("O fatorial do número é "+
28                            fatorial(num));
29
30          } // fim do metodo main
31
32    } // fim da classe FatorialRecursivo
```

Programa 22: Cálculo do fatorial de um número utilizando um método recursivo.

```
fatorial(5)
↓              valor retornado: 5 * 24 = 120
5 * fatorial(4)
    ↓              valor retornado: 4 * 6 = 24
    4 * fatorial(3)
        ↓              valor retornado: 3 * 2 = 6
        3 * fatorial(2)
            ↓              valor retornado: 2 * 1 = 2
            2 * fatorial(1)
                ↓              valor retornado: 1
                1
```

Figura 26: Cálculo recursivo do fatorial de 5.

A Figura 26 mostra como é processado o cálculo do fatorial de 5 pelo método recursivo. Note que a cada chamada do método *fatorial*, dependendo do valor passado em *n*, uma nova chamada recursiva pode ocorrer. Cada chamada recursiva é independente da outra, ou seja, quando uma nova chamada recursiva de *fatorial* é feita, as anteriores continuam ativas. Isso quer dizer que, no exemplo da Figura 26, quando é feita a chamada *fatorial(1)* já existem 4 chamadas ativas do método.

O que foi explicado no parágrafo anterior nos leva a uma conclusão importante. A seqüência de chamadas recursivas deve parar, caso contrário a execução do método nunca terminará. Quem controla essa parada é a implementação do caso básico. Quando o valor de *n* chegar a um valor menor ou igual a 1, o método simplesmente retorna o valor 1, sem fazer novas chamadas recursivas. Quando isso ocorre inicia-se uma seqüência de retornos que segue para cima, até a chamada original, que devolve o resultado final do cálculo. Logo, toda seqüência de chamadas recursivas deve sempre convergir para o caso básico.

Você pode pensar: qual a vantagem de se utilizar a recursão? Não poderia implementar o cálculo do fatorial utilizando um laço *for* ou *while*?

A solução que utiliza um laço chama-se solução *iterativa*. Tanto a iteração quanto a recursão envolvem repetição: a iteração utiliza explicitamente uma estrutura de repetição enquanto a recursão faz isso através de chamadas repetidas de métodos. A iteração e a recursão envolvem um teste de terminação: a iteração termina quando a condição do laço deixa de ser satisfeita e a recursão quando se atinge o caso básico. Tanto a iteração quanto a recursão podem ocorrer infinitamente: um laço infinito ocorre quando a condição nunca se torna falso; a recursão infinita ocorre quando o caso básico nunca é atingido.

A recursão tem pontos negativos. Ela invoca repetidamente o mecanismo de chamadas de método, o que pode ser caro em tempo de processador e espaço de memória. Cada chamada recursiva faz com que outra cópia do método seja criada. Esse conjunto de cópias pode consumir um espaço de memória considerável, caso o método possua parâmetros e variáveis locais em grande quantidade ou que sejam grandes. A iteração normalmente ocorre dentro de um método, de modo que a sobrecarga das chamadas de método repetidas e a atribuição extra de memória são evitadas. Por que, então, escolher a recursão?

Qualquer problema resolvido iterativamente pode ser resolvido recursivamente e vice-versa. Em geral, uma abordagem recursiva é escolhida preferencialmente quando a abordagem recursiva quando torna a solução do problema mais fácil. Geralmente, uma abordagem recursiva pode ser implementada com poucas linhas de código e uma abordagem iterativa correspondente pode exigir quantidades maiores de código.

Deve-se evitar o uso da recursão quando o desempenho e uso de memória são fatores críticos no problema. As chamadas recursivas, como vimos, levam tempo e consomem memória adicional.

12.2. Sobrecarga

A linguagem Java permite que se defina vários métodos com o mesmo nome, contanto que eles tenham conjuntos diferentes de parâmetros. Esse recurso é conhecido como *sobrecarga de método*. Quando se chama um método sobrecarregado, o compilador Java seleciona o método adequado examinando a quantidade, os tipos e a ordem dos argumentos na chamada. A sobrecarga de métodos é comum quando se deseja criar vários métodos com o mesmo nome que realizam tarefas semelhantes, mas sobre tipos de dados diferentes.

O Programa 23 ilustra o uso de sobrecarga criando dois métodos para cálculo do quadrado de um número: um que trabalha com números inteiros e outro que trabalha com números reais. Veja na Figura 27 o resultado da chamada dos métodos para os valores 5 e 2.5. Sempre que o método *quadrado* for chamado com um número inteiro, o resultado será inteiro e quando for chamado com um valor do tipo *double*, o resultado será do tipo *double*.

```
1     // Programa 23: QuadradoComSobrecarga.java
2     // Programa que faz uso de um metodo sobrecarregado para calcular
3     // o quadrado de um numero inteiro ou real.
4
5     public class QuadradoComSobrecarga {
6
7           public static int quadrado( int n ) {
8
9                 return n * n;
10
11          } // fim do metodo quadrado com parametro int
12
13          public static double quadrado( double n ) {
14
15                return n * n;
16
17          } // fim do metodo quadrado com parametro double
18
19          public static void main(String[] args)
20          {
21                System.out.println(quadrado(5));
22
23                System.out.println(quadrado(2.5));
24
25          } // fim do metodo main
26
27    } // fim da classe QuadradoComSobrecarga
```

Programa 23: Exemplo de sobrecarga de um método que faz o cálculo do quadrado de um número.

```
25
6.25
```

Figura 27: Saída mostrada pelo programa do cálculo do quadrado utilizando métodos sobrecarregados (Programa 23).

Cuidado: o compilador Java não diferenciará métodos que diferem apenas pelo tipo de retorno, porque os métodos só podem ser distinguidos pelo tipo e ordem de seus parâmetros. Criar métodos sobrecarregados com listas de parâmetros idênticas e tipos de retorno diferentes é um erro de sintaxe.

12.3. Definição de Métodos Estáticos em Classes Separadas

Até o momento, definimos todos os métodos que utilizamos em nosso programa dentro da mesma classe do mesmo. Mas existe a possibilidade de definirmos métodos estáticos fora da classe do programa, através da definição de outras classes. Por exemplo, a Figura 28 mostra a definição de

uma classe chamada *Matematica*, que define três métodos: *quadrado* (sobrecarregado), *potInt* e *fatorial*.

```
1     // Figura 28: Matematica.java
2     // Esta classe define alguns metodos uteis para calculos
3     // matematicos.
4
5     public class Matematica {
6
7           public static int quadrado( int n ) {
8
9                 return n * n;
10
11          } // fim do metodo quadrado com parametro int
12
13          public static double quadrado( double n ) {
14
15                return n * n;
16
17          } // fim do metodo quadrado com parametro double
18
19          public static double potInt(double base, int expoente)
20          {
21                double result = 1;
22
23                for(; expoente >= 1; expoente--)
24                      result *= base;
25
26                return result;
27
28          } // fim do metodo potInt
29
30          public static int fatorial(int n)
31          {
32                if(n <= 1)
33                      return 1;
34                else
35                      return n * fatorial( n - 1 );
36
37          } // fim do metodo fatorial
38
39    } // fim da classe Matematica
```

Figura 28: Classe que implementa alguns métodos matemáticos estáticos.

Esta classe não é um programa. Isso porque não está definido o método *main*. Essa classe pode e deve ser compilada, mas não pode ser executada. Para que serve esta classe então? Na verdade ela será útil, pois outros programas podem utilizá-la, fazendo uso de seus métodos. Por exemplo, o Programa 24 faz uso dos métodos da classe *Matematica* para fazer alguns cálculos com números. Note que como os métodos utilizados pertencem a uma classe externa, o nome da classe deve ser incluído no nome do método (assim como fizemos com os métodos da classe *Math*).

```
1     // Programa 24: TesteMatematica.java
2     // Programa que faz uso dos metodos da classe Matematica para
3     // realizar calculos com numeros.
4
5     public class TesteMatematica {
6
7           public static void main(String[] args)
8           {
9                 System.out.println("7.5 ao quadrado eh " +
10                      Matematica.quadrado(7.5));
11
12                System.out.println("12 ao quadrado eh " +
13                      Matematica.quadrado(12));
14
15                System.out.println("O fatorial de 10 eh " +
16                      Matematica.fatorial(10));
17
18                System.out.println("2.5 elevado na 6 eh " +
19                      Matematica.potInt(2.5, 6));
20
21          } // fim do metodo main
22
23    } // fim da classe TesteMatematica
```

Programa 24: Programa que faz uso da classe Matematica.

Uma observação importante: para que um programa utilize os métodos da classe *Matematica*, o arquivo *Matematica.class* deve estar no mesmo diretório. Caso contrário, ocorrerá um erro durante a compilação, pois o compilador Java não será capaz de encontrar a classe. O mesmo vale para a execução do programa.

Até agora não sabemos para que serve a palavra *public* na frente do nome do método. Pois bem, a palavra *public* serve para dizer ao compilador Java que o método definido pode ser utilizado por outras classes externas. Nós conseguimos utilizar os métodos da classe *Matematica* dentro da classe *TesteMatematica* porque todos eles são *public*.

Quando um método não é público, ele pode ser *privado*. Um método privado só pode ser utilizado dentro da classe em que ele é definido. Especificamos que um método é privado quando trocamos a palavra *public* por *private*. Por exemplo, se o método *fatorial* da classe *Matematica* fosse definido como privado, ele não poderia ser utilizado dentro da classe *TesteMatematica*, mas única e exclusivamente por métodos da classe *Matematica*.

Métodos privados podem ser úteis quando desejamos dividir a implementação de um método complexo em diversos métodos menores, mas disponibilizar apenas o método complexo para uso por outras classes. Discutiremos mais sobre o uso de métodos privados quando estudarmos a programação baseada em objetos.

12.4. Exercícios

1. Escreva um método recursivo *potencia (base, expoente)* que, quando chamado, retorna *base* elevado a *expoente*. Suponha que expoente é um inteiro maior ou igual a 0. Incorpore esse método a um programa em Java que permite ao usuário digitar a base e o expoente e calcular a potência.

2. O máximo divisor comum dos inteiros x e y é o maior inteiro que divide x e y. Escreva um método recursivo chamado gcd que retorna o máximo divisor comum de x e y. O gcd de x e y é definido recursivamente como segue: se y for igual a 0, então gcd(x,y) é x; caso contrário, gcd(x,y) é gcd(y, x mod y). Utilize esse método em um programa que calcula o máximo divisor comum para 5 pares de números digitados pelo usuário.

3. O que faz o seguinte método?

```
// o parâmetro b deve ser um inteiro
// positivo para evitar recursão infinita
public int mystery( int a, int b )
{
      if(b == 1)
            return a;
      else
            return a + mystery( a, b - 1 );
}
```

4. Localize o erro no seguinte método recursivo e explique como corrigi-lo:

```
public int sum( int n )
{
      if( n == 0 )
            return 0;
      else
            return n + sum( n );
}
```

5. Escreva um método recursivo que retorne o n-ésimo termo da seqüência de Fibonacci. O n-ésimo termo da seqüência de Fibonacci é obtido pela soma do termo *n – 1* mais termo *n – 2*. Utilize como caso básico o fato de que os termos de número 1 e 2 valem 1.

6. O que é mostrado pelo programa abaixo? Justifique sua resposta.

```
public class Mystery2 {

      public static void linha(int tam) {
            for(int i = tam; i > 0; i++ ) {
                  System.out.println("*");
            }
      }

      public static void linha(double tam) {
            for(int i = 0; i <= tam/2; i++ ) {
                  System.out.print("%");
            }
            for(int i = 0; i <= tam/2; i++ ) {
                  System.out.print("-");
            }
            System.out.println();
      }

      public static void main(String[] args) {
            linha(10.0);
            linha(5);
            linha(6.0);
            linha( 18 / 4 );
      }
}
```

7. Escreva um método recursivo que calcule a soma dos números inteiros de 1 a N, sendo N um valor inteiro positivo. Incorpore esse método a um programa que leia o valor de N do usuário e mostre o valor da soma dos números de 1 a N.

13. Estruturas Homogêneas de Dados

Os tipos de dados que vimos até aqui são capazes de armazenar apenas um único valor de cada vez. Por isso, são chamados *Tipos Simples de Dados*, e as variáveis definidas com estes tipos de dados chamadas *Variáveis Simples*. Estes tipos de dados cobrem grande parte das necessidades de armazenamento de dados dos programas.

Entretanto, ocorrem situações em que o programador precisa armazenar, em seqüência, diversos valores de um mesmo tipo e acessá-los utilizando um índice numérico. Por exemplo, como armazenaríamos, em um programa, uma lista ordenada dos aprovados no teste de classificação do CTI? Supondo que estivéssemos trabalhando com uma listagem de 5 pessoas, poderíamos utilizar cinco variáveis do tipo literal, como abaixo:

```
var
      Candidato1, Candidato2,
      Candidato3, Candidato4, Candidato5 : literal;
```

Poderíamos armazenar o primeiro colocado na variável *Candidato1*, o segundo na variável *Candidato2*, e assim por diante. Mas e se tivéssemos em vez de cinco, trinta candidatos. Será que teríamos que criar trinta variáveis do tipo literal? Isto tornaria a programação no mínimo cansativa.

Para contornar esse problema, existem as chamadas *Variáveis Indexadas*, conhecidas também como *Variáveis Subscritas* ou *Arranjos*. Este tipo de variável constitui o que se denominam *Estruturas Homogêneas*, uma vez que todos os valores indexados devem ser de um mesmo tipo de dado. Cada dado é acessado especificando-se o número de seu índice, ou seja, a posição em que se encontra na seqüência de dados do arranjo.

Podemos comparar um arranjo com um prédio residencial. Um prédio contém um conjunto de apartamentos, geralmente todos do mesmo tipo (residenciais). Um arranjo conterá um conjunto de dados, todos do mesmo tipo. É possível termos um arranjo de inteiros, de literais, etc, mas não um arranjo com dados metade inteiros e metade literais, por exemplo.

Quando desejamos visitar um determinado morador do prédio, precisamos conhecer o número do apartamento em que mora. Todos os apartamentos do prédio possuem um número, que é diferente dos demais. O mesmo ocorre com os arranjos. Cada dado armazenado está guardado em uma posição (apartamento) diferente. Quando desejamos consultar o valor de uma determinada posição, precisamos especificar o seu número. Cada posição possui um número diferente.

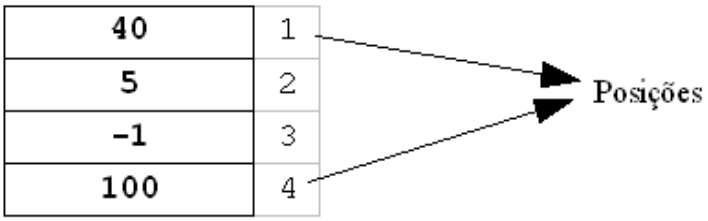


Figura 29: Representação de um arranjo de 5 posições mostrando os dados armazenados (números a esquerda) e suas respectivas posições dentro do arranjo (números a direita).

Cada posição de um arranjo se comporta como uma variável simples. Tudo o que fazemos com uma variável simples, podemos fazer com uma posição de um arranjo. A Figura 29 ilustra um arranjo de inteiros contendo 5 posições.

13.1. Declaração de Arranjos

A declaração de arranjos é semelhante à declaração de variáveis simples. A diferença aqui é que devemos especificar a quantidade de elementos que o arranjo será capaz de armazenar. O tipo de dado também deve ser explicado na declaração. O formato da declaração de um arranjo é o seguinte:

```
VAR nome_da_variável : ARRANJO [ pos_ini .. pos_fim ] DE tipo;
```

Onde:

- *var* é a palavra-chave para o início da declaração de variáveis. Como arranjos são variáveis, devem ser declarados nesta parte dos algoritmos;
- *nome_da_variável* é o nome da variável que conterá o arranjo. Este nome segue as mesmas regras dos nomes de variáveis simples;
- *arranjo* é uma palavra-chave para especificar que se está declarando um arranjo;
- *pos_ini* e *pos_fim* especificam o primeiro e último índices (número de posições) da seqüência de dados que será armazenada no arranjo. Estes dois números devem ser separados por dois pontos finais "..". Os números de cada posição aumentaram de 1 em 1, a partir de *pos_ini* até chegar em *pos_fim*;
- *tipo* é o tipo dos dados que serão armazenados no arranjo.

Como exemplo , vamos declarar um arranjo para armazenar a listagem de candidatos aprovados no teste de classificação, considerando 30 aprovados:

```
var Aprovados : arranjo[1..30] de literal;
```

Cada posição do arranjo *Aprovados* é capaz de armazenar um dado literal. Este arranjo pode ser comparado a um engradado de garrafas, com espaço para 30 garrafas. Cada espaço é numerado com um número diferente, de 1 a 30. A primeira é representada pelo número 1, a segunda pelo número 2 e a última pelo número 30.

Para acessarmos o 2º colocado no concurso, bastaria consultarmos a posição 2 do arranjo, escrevendo *Aprovados[2]*. Note que para fazermos referência a uma posição do arranjo, basta escrevermos o nome da variável seguida pelo número da posição entre colchetes.

Um outro exemplo: imagine que declaremos o arranjo abaixo:

```
var Conjunto : arranjo [0..19] de real;
```

O arranjo *Conjunto* declarado acima possui 20 posições, sendo que cada uma pode guardar um número real. A primeira posição é representada pelo número zero, a terceira pelo número 2 e a última pelo número 19. Para acessarmos a 10ª posição do arranjo, basta escrevermos *Conjunto[9]*. Usamos 9 e não 10 porque devemos contar 10 posições a partir do número zero, o número da primeira posição.

Os dois arranjos que declaramos anteriormente são conhecidos como *arranjos unidimensionais* ou *vetores*. Arranjos desse tipo possuem apenas uma dimensão, pois seus elementos podem ser dispostos sobre uma linha, como mostrado na Figura 30.

José	Maria	João	Pedro	Leonardo	...
Aprovados[1]	Aprovados[2]	Aprovados[3]	Aprovados[4]	Aprovados[5]	

Figura 30: Representação linear de um vetor.

Vamos a um exemplo prático do uso de vetores. O Algoritmo 31 cria um vetor de 20 posições chamado *numeros*, armazena os números de 1 a 20 dentro dele, e mostra os números na tela.

```
Algoritmo TesteVetor;
var
     numeros : arranjo [1..20] de inteiro;
     pos : inteiro;
inicio
    para pos de 1 até 20 incr de 1 faça
          numeros[pos] ← pos;
     fim para;

     escreva "Numeros de 1 a 20: ";
     para pos de 1 até 20 incr de 1 faça
          escreva numeros[pos];
     fim para;
fim.
```

Algoritmo 31: Algoritmo que preenche um vetor de 20 elementos e mostra seu conteúdo na tela.

O número de posição entre colchetes deve ser um valor inteiro, uma variável ou constante do tipo inteiro, ou uma expressão de resultado inteiro. Se o algoritmo utilizar uma expressão como o índice (o número entre colchetes), ela será avaliada para se obter o número a posição desejada. Por exemplo, se assumirmos que a variável *a* vale 5 e que a variável *b* vale 6, então a instrução

```
numeros[ a + b ] ← 10;
```

atribui o valor 10 à posição de número 11 do vetor *numeros*.

13.2. Exemplos de Algoritmos Utilizando Vetores

13.2.1. Somando os Elementos de um Vetor

Freqüentemente os elementos de um vetor representam uma série de valores que devem ser usados em um cálculo. Por exemplo, se os elementos de um vetor representam as alturas, em centímetros, de várias pessoas, é possível calcular a média de alturas dessas pessoas e verificar quantas delas possuem altura maior do que a média.

O Algoritmo 32 faz a leitura de alturas de 10 pessoas e armazena essas alturas no vetor *alturas*. Logo a seguir é feita a leitura das 10 alturas na estrutura *para...faça* subseqüente. Na segunda estrutura *para...faça* é calculada a soma das alturas. Após, é calculada a média e a última estrutura *para...faça* se contam quantas pessoas possuem altura superior à média.

```
Algoritmo MediaAlturas;
var
      alturas : arranjo [1..10] de inteiro;
      cont, soma, maiores : inteiro;
      media : real;

inicio
     para cont de 1 até 10 incr de 1 faça
           escreva "Digite a altura da "+cont+"ª pessoa: ";
           leia alturas[cont];
      fim para;

      soma ← 0;
      para cont de 1 até 10 incr de 1 faça
           soma ← soma + alturas[cont];
      fim para;

      media ← soma / 10.0;

      maiores ← 0;
      para cont de 1 até 10 incr de 1 faça
           se alturas[cont] > media então
                maiores ← maiores + 1;
           fim se;
      fim para;

      escreva maiores + " pessoas possuem altura maior do que a média";
fim.
```

Algoritmo 32: Calcula a média de alturas de 10 pessoas e verifica quantas delas possuem altura maior do que a média.

13.2.2. Utilizando os Elementos de um Vetor como Contadores

Algumas vezes os programas usam uma série de variáveis contadoras para resumir dados, como os resultados de uma pesquisa. Imagine o seguinte problema: foi feita uma pesquisa na rua sobre a audiência de alguns canais de televisão. Nesta pesquisa, cada pessoa respondeu qual o número do canal que mais assistia diariamente (um número de 1 a 10).

O problema é escrever um algoritmo que, a partir dos dados da pesquisa (uma lista de respostas de 100 pessoas) mostre o número de pessoas que escolheram cada canal. Com o que

conhecíamos antes, seria necessário criar 10 variáveis para contar a quantidade de pessoas que escolheram cada canal. Mas com o uso de vetores, podemos resolver isso com uma única variável!

O Algoritmo 33 mostra a solução do problema utilizando um vetor. O primeiro *para...faça* inicializa todas as posições do vetor *contadores* com zero. Isso é necessário, pois cada uma será utilizada como um contador. O segundo *para...faça* faz a leitura das respostas de cada pessoa (cada resposta é um número inteiro representando o canal escolhido). Note que o próprio número do canal, armazenado na variável *canal*, é utilizado como índice para incrementar a respectiva posição no vetor. O último *para...faça* mostra os resultados da pesquisa.

```
Algoritmo PesquisaTV;
var
     contadores : arranjo [1..10] de inteiro;
     cont, canal : inteiro;
inicio
     para cont de 1 até 10 incr de 1 faça
          contadores[cont] ← 0;
     fim para;

     para cont de 1 até 100 incr de 1 faça
          escreva "Digite a resposta da "+cont+"ª pessoa: ";
          leia canal;
          contadores[canal] ← contadores[canal] + 1;
     fim para;

     para cont de 1 até 10 incr de 1 faça
          escreva contadores[cont] + " assistem o canal " + cont;
     fim para;
fim.
```

Algoritmo 33: Algoritmo que totaliza os resultados de uma pesquisa de audiência para 10 canais de TV.

13.3. Uso de Arranjos em Java

A linguagem Java, como a grande maioria das linguagens de programação de alto nível, permite que trabalhemos com arranjos. Assim como vimos para os algoritmos, todo arranjo em Java será uma variável, e como toda variável precisa ser declarado. A declaração de um arranjo em Java é feita do seguinte modo:

```
tipo nome_da_variavel[] = new tipo [ tamanho ];
```

que também pode ser escrita em duas etapas, como segue:

```
tipo nome_da_variavel[];
nome_da_variavel = new tipo [ tamanho ];
```

Por exemplo, para declararmos um arranjo *numeros*, com 20 posições, cada uma sendo capaz de

armazenar um valor do tipo *int*, escrevemos:

```
int numeros[] = new int[20];
```

ou

```
int numeros[];
numeros = new int[20];
```

O operador *new*, utilizado na declaração do vetor acima, serve para a alocação do espaço de memória para o arranjo. Alocar significa reservar, ou seja, aquele espaço de memória não poderá ser utilizado por outra variável ou arranjo. Todas as variáveis que declaramos com tipos primitivos são alocadas automaticamente. Arranjos em Java não. Por isso, é necessário o uso do operador *new*, pois um arranjo só poderá ser utilizado após sua alocação.

Note que ao contrário dos algoritmos, não especificamos em Java a posição inicial e a posição final, mas somente o número de elementos. Em Java, o primeiro elemento sempre é representado pelo número zero. Logo, a declaração acima em algoritmos seria feita da seguinte forma:

```
var numeros : arranjo [0..19] de inteiro;
```

O último elemento sempre será representado pelo índice equivalente ao número de elementos do arranjo menos 1. Por exemplo, no exemplo acima, a última posição do vetor *numeros* será 19 e não 20.

Ao declarar um arranjo, o tipo dos elementos e os colchetes podem ser combinados no início da declaração para indicar que todos os identificadores na declaração representam arranjos, como em

```
double[] array1, array2;
```

que declara *array1* e *array2* como arranjos do tipo *double*.

O Programa 25 acesso a elementos de um arranjo em Java é feita da mesma forma que nos algoritmos, escrevendo-se o nome da variável seguida pelo número da posição desejada entre colchetes. O mostra a implementação em Java do Algoritmo 31.

Vamos analisar o primeiro *for*. Este for coloca, em cada posição do vetor *numeros*, um número diferente de 1 a 20. A variável *pos* varia de 1 a 20. Como as posições do vetor são numeradas de 0 a 19 (pois o vetor foi alocado com 20 elementos), o número da posição foi calculado pela expressão *pos – 1*. Isso foi necessário, pois se deixássemos o acesso com o valor armazenado na variável *pos* aconteceria que, quando o for atingisse o número 20, o programa terminaria com um erro, pois foi acessada uma posição inválida do vetor. Uma solução semelhante foi feita no segundo *for*, utilizado para mostrar o conteúdo do vetor.

Existe uma outra forma de se alocar arranjos em Java. Esta segunda forma nos permite alocar e inicializar os elementos do arranjo de uma só vez. Para isso, basta escrevermos um sinal de igual e uma *lista de inicializadores* separados por vírgulas e colocados entre chaves, como mostrado abaixo:

```
int n[] = {10, 20, 30, 40, 50};
```

Essa instrução declarará um arranjo *n* de elementos do tipo *in*t, com a primeira posição contendo o valor 10, a segunda contendo o valor 20, a terceira contendo o valor 30, e assim por diante. O

tamanho do vetor será determinado pelo número de elementos na lista de inicializadores, no caso acima 5. Observe que a declaração acima não precisa do operador *new* para alocar o vetor.

```
1     // Programa 25: TesteVetor.java
2     // Programa que ilustra os conceitos basicos de arranjos
3
4     public class TesteVetor {
5
6           public static void main(String[] args)
7           {
8                 int[] numeros = new int[20];
9
10                for(int pos = 1; pos <= 20; pos++)
11                      numeros[pos - 1] = pos;
12
13                System.out.println("Numeros de 1 a 20: ");
14
15                for(int pos = 1; pos <= 20; pos++)
16                      System.out.println(numeros[pos - 1]);
17
18          } // fim do metodo main
19
20    } // fim da classe TesteVetor
```

Programa 25: Programa que ilustra os conceitos básicos de arranjos em Java (implementação do Algoritmo 31).

O Programa 26 mostra uma reimplementação do Programa 25 utilizando uma lista de inicializadores para criar o vetor *numeros*. Note que o primeiro for que utilizamos a primeira solução não foi mais necessário.

```
1     // Programa 26: TesteVetor2.java
2     // Programa que ilustra o uso da lista de inicializadores
3
4     public class TesteVetor2 {
5
6           public static void main(String[] args)
7           {
8                 int[] numeros = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
9                                   11, 12, 13, 14, 15, 16, 17, 18, 19,
10                                  20 };
11
12                for(int pos = 1; pos <= 20; pos++)
13                      System.out.println(numeros[pos - 1]);
14
15          } // fim do metodo main
16
17    } // fim da classe TesteVetor2
```

Programa 26: Programa que mostra os números de 1 a 20 utilizando uma lista de inicializadores.

13.4. Referências e Parâmetros por Referência

Duas maneiras de passar parâmetros para métodos em muitas linguagens de programação são a *passagem por valor* e a *passagem por referência*. A passagem por valor é a que vimos até agora, quando se faz uma cópia do valor do parâmetro e passa-se para o método. Alterações na cópia feitas pelo método chamado não afetam o valor da variável original no código chamador. Por exemplo, considere o método abaixo:

```
public static void proc(int y) {
      y++;
      System.out.println("Durante Y = " + y);
}
```

O parâmetro *y*, passado para o método *proc*, é passado por valor. Isso significa que alterações no valor de *y* não serão propagadas para fora do método. Veja que *y* é alterado dentro do método. Um exemplo de utilização do método acima é mostrado abaixo. A saída do código é mostrada na Figura 31. Note que o valor da variável *x* não é alterado pelo método *proc*. O parâmetro *y* recebe uma cópia do valor da variável *x*.

```
int x = 1;
System.out.println("Antes X = " + x);
proc(x);
System.out.println("Depois X = " + x);
```

```
Antes X = 1
Durante Y = 2
Depois X = 1
```

Figura 31: Saída do código que usa passagem por valor.

Quando um parâmetro é passado por referência, o chamador dá ao método a capacidade de acessar diretamente os dados do chamador e modificar esses dados se o método chamado assim o escolher. A passagem por referência melhora o desempenho para a passagem de quantidades grandes de dados, porque ela elimina a necessidade de se construir uma cópia dos dados.

Diferentemente de outras linguagens, Java não permite que o programador escolha entre passar cada parâmetro por valor ou por referência. As variáveis dos tipos de dados primitivos sempre são passadas por valor. Variáveis que armazenam objetos (veremos mais sobre eles mais tarde) ou arranjos são sempre passadas por referência. Veremos um exemplo de passagem por referência na próxima seção.

Passar arranjos por referência faz sentido por razões de desempenho. Se os arranjos fossem passados por valor, uma cópia de cada elemento seria passada. Para os arranjos grandes passados com freqüência, isso desperdiçaria tempo e consumiria considerável capacidade de armazenamento para as cópias dos dados.

13.5. Passando Arranjos para Métodos

Para passar um arranjo como argumento para um método, especifique o nome do arranjo sem nenhum colchete. Por exemplo, se o arranjo *temperaturasPorDia* é declarado como

```
int temperaturasPorDia[] = new int[30];
```

a chamada de método

```
modificaArranjo( temperaturasPorDia );
```

passa o arranjo *temperaturasPorDia* para o método *modificaArranjo*.

Para um método receber um arranjo como parâmetro, a lista de parâmetros dele deve conter uma especificação de arranjo (ou várias, se mais de um arranjo deve ser recebido). Por exemplo, o cabeçalho do método *modificaArranjo* poderia ser escrito assim:

```
void modificaArranjo( int b[] )
```

indicando que o método espera receber um arranjo de inteiros no parâmetro *b*.

O Programa 27 ilustra a passagem de arranjos para métodos. A saída do programa é mostrada na Figura 32.

```
1    // Programa 27: PassagemPorReferencia.java
2    // Programa que ilustra a passagem de arranjos para metodos.
3
4    public class PassagemPorReferencia {
5
6          public static void modificaArranjo( int b[] )
7          {
8                for(int cont = 0; cont < b.length; cont++ )
9                      b[cont] *= 2;
10         }
11
12         public static void main(String[] args)
13         {
14               int[] numeros = { 1, 2, 3, 4, 5 };
15
16               System.out.print("Valores originais: ");
17
18               for(int pos = 0; pos < 5; pos++)
19                     System.out.println(numeros[pos] + " ");
20
21               modificaArranjo( numeros );
22
23               System.out.print("Valores modificados: ");
24
25               for(int pos = 0; pos < 5; pos++)
26                     System.out.println(numeros[pos] + " ");
27
28         } // fim do metodo main
29
30   } // fim da classe PassagemPorReferencia
```

Programa 27: Demonstração da passagem de um arranjo para um método em Java.

```
Valores originais...: 1 2 3 4 5
Valores modificados.: 2 4 6 8 10
```

Figura 32: Saída do Programa 27.

Note que, após a chamada ao método *modificaArranjo*, o conteúdo da variável *numeros* foi modificado. Isso aconteceu, porque o arranjo foi passado por referência. Todas as modificações feitas no parâmetro *b* foram também realizadas em *numeros*. Note também a linha 8. Nessa linha é utilizado um atributo especial de todos os arranjos em Java chamado *length*. Esse atributo contém o número de elementos do arranjo. No caso do exemplo, a chamada devolveria 5, o número de elementos do arranjo *numeros*. Para fazer uso do atributo *length*, basta escrever o nome da variável que contém o arranjo seguido de um ponto final e da palavra *length*.

13.6. Exercícios

1. Localize o erro em cada um dos seguintes segmentos de programa e corrija o erro.

```
a) int b[] = new int[10];
   for( int i = 0; i <= b.length; i++ )
       b[i] = 1;
b) int a[];
   a[0] = 10;
   System.out.println(a[0]);
```

2. Implemente em Java o Algoritmo 32.

3. Implemente em Java o Algoritmo 33.

4. Escreva um programa em Java que leia uma seqüência de N números inteiros e mostre essa seqüência ao contrário, no final de sua execução. O valor de N deve ser informado pelo usuário.

5. Escreva um programa em Java que leia a pontuação de N times de um campeonato de futebol. O programa deve mostrar, ao final, qual a pontuação dos campeões, bem como mostrar quantos times dividiram o título.

6. Escreva um programa em Java que simule o lançamento de dois dados. O programa deve lançar dois dados, sorteando um número aleatório entre 1 e 6 para cada um. A soma dos dois valores deve então ser calculada (como cada dado pode mostrar um valor inteiro de 1 a 6, a soma dos valores irá variar de 2 a 12). O programa deve lançar os dados 36000 vezes e contar quantas vezes cada soma ocorreu.

14. Introdução à Programação Orientada a Objetos

Até este momento, temos desenvolvido nossas aplicações com base em dois métodos: a *Programação Estruturada* e a *Programação Modular*. No uso da programação estruturada, aprendemos a utilizar as estruturas de controle de forma adequada na construção de algoritmos que solucionem determinados problemas. Já na programação modular, aprendemos a dividir nossos algoritmos em módulos, constituídos de procedimentos e funções (ou no caso de Java, de métodos) que utilizados juntos, resolvem o problema como um todo. Neste capítulo final, aprenderemos a utilizar uma nova abordagem: a *Programação Orientada a Objetos*.

A programação orientada a objetos (também conhecida como *OOP*, do inglês *Object-Oriented Programming*) pode ser considerada como uma extensão da programação modular. Entretanto, sistemas criados sob essa tecnologia são muito mais funcionais, mais consistentes e mais robustos do que aqueles criados no paradigma da programação modular. Isto se deve ao fato de que sistemas produzidos na tecnologia orientada a objetos são baseados numa estrutura ativa e dinâmica: os *Objetos.* Assim, enquanto a programação modular estrutura o programa baseando-se em procedimentos, funções e dados (separados um do outro) e escondendo as estruturas de dados "dentro" das rotinas, a OOP o faz encapsulando os dados e rotinas nos objetos.

Mas o que é, afinal de contas um objeto? Bom, desde crianças, formulamos esse conceito. Sabemos que um objeto é qualquer coisa que possa ser manipulada por uma pessoa ou outro agente. Sabemos que um cadeira, um copo, uma caneta ou um CD são exemplos de objetos.

Dentro da área de programação de computadores, o termo *objeto* tem um sentido similar. Olhando por outro ângulo, podemos ver que os objetos do mundo real têm propriedades ou atributos e podemos interagir com eles. Os objetos virtuais, que trabalharemos em nossos programas, possuirão, também, atributos e também permitirão interação com o código do programa e até mesmo com outros objetos. Nas próximas sessões começaremos a aprender, passo a passo, como construir cada parte de um objeto.

14.1. Classes e Atributos de um Objeto

Todos os objetos possuem atributos. Por exemplo, uma cadeira pode possuir três atributos: *cor*, *tipo de material* (madeira, metal, etc) e *tamanho*. Um carro possui outras propriedades, por exemplo: *cor*, *proprietário*, *placa*, *ano*, *fabricante*, etc. Assim como esses exemplos, cada objeto usado em nossos programas terá um conjunto de atributos.

Os objetos do mundo real não são construídos ao acaso. Todos são construídos a partir de um projeto, de uma planta em papel. Em outras palavras, objetos que pertençam ao mesmo *tipo*, tem os mesmos tipos de atributos, diferindo no valor deles. Por exemplo, todos os carros do tipo Fusca são semelhantes e possuem os mesmos atributos: *cor*, *ano*, *proprietário*. O que vai mudar são os valores dos atributos para cada fusca individual. Por exemplo, a Tabela 15 mostra três exemplos de fuscas, cada um com um valor diferente para as propriedades.

Cor	***Ano***	***Proprietário***
Azul	1960	Fulano de Tal
Amarelo	1970	Josicrêidson das Neves
Vermelho	1972	Florentina Souza

Tabela 15: Objetos do mesmo tipo, com os valores particulares dos atributos.

Vemos que o que diferencia objetos do mesmo tipo não são os nomes dos atributos que cada um possui, mas os *valores* desses atributos. Logo, podemos definir um projeto, uma planta, a partir da qual construir nossos objetos.

Em nossos programas, essa planta básica de objetos se chama *Classe*. Uma classe é uma implementação, em um programa, de um tipo de objeto. É na classe que especificaremos a definição dos atributos que cada objeto construído a partir dela terá.

Especificaremos nossas classes em papel, através de uma notação modificada, baseada em *UML* (*Unified Modeling Language*). Em nossa notação, as classes serão representadas como mostrado na Figura 32. O *nome da classe* é um nome qualquer, seguindo as regras de nomenclatura que já utilizamos para as variáveis. Nomes de classe geralmente começam com letra maiúscula. Os atributos serão especificados como uma lista, um abaixo do outro. A especificação dos métodos veremos mais tarde, deixando em branco por enquanto.

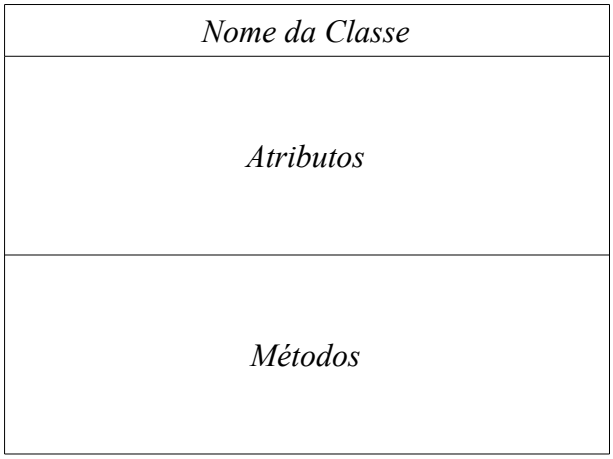


Figura 33: Estrutura básica da notação de classes que utilizaremos.

Como exemplo, vamos especificar uma classe para representar os objetos Fusca. A especificação é mostrada na Figura 32. O nome da nossa classe é *Fusca*. Cada atributo deve ser especificado e para cada um será definido um nome e um tipo.

Cada classe, em termos de algoritmos, é um novo tipo de dados. Ou seja, podemos declarar uma variável, por exemplo, do tipo *Fusca*. Definir classes nos possibilita criar tipos de dados próprios, além dos já conhecidos tipos primitivos: inteiro, literal, caractere, real e lógico.

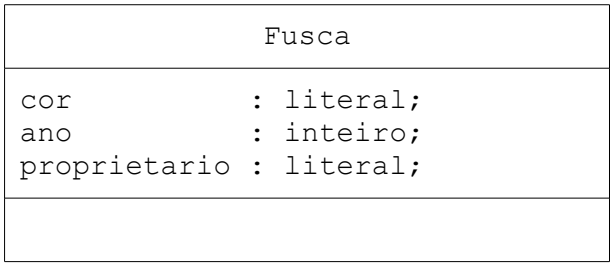


Figura 34: Representação da classe Fusca em nossa notação.

A classe definida acima pode ser implementada em Java, como mostra a Figura 35. A palavra *classe* não é nenhuma novidade para nós. Todo o programa que construímos e todo método definido por nós até agora estava dentro de uma classe. Definimos classes, mas não havíamos compreendido ainda o seu real significado. Como vimos, cada classe em Java deve estar definida em um arquivo *.java* diferente, nomeado com o mesmo nome da classe.

Os atributos de uma classe são definidos como se fossem variáveis. Escrevemos seu tipo e após, o nome do atributo. À frente de tudo isso, devemos especificar o tipo de acesso do atributo: *public* ou *private*. Por enquanto, todos os atributos que definirmos serão do tipo *public*.

```
1     // Figura 35: Fusca.java
2     // Classe que contem a estrutura de todos os objetos Fusca.
3
4     public class Fusca {
5
6           public String cor;
7
8           public int ano;
9
10          public String proprietario;
11
12    } // fim da classe Fusca
```

Figura 35: Implementação da classe Fusca em Java.

O Programa 28 ilustra o uso da classe *Fusca*. Uma vez que a classe foi definida, ela pode ser utilizada em declarações como as mostradas na linha 8 do programa

```
Fusca f1, f2, f3;
```

O nome da classe é um novo tipo de dados. Pode haver muitas variáveis de uma classe, da mesma forma que pode haver muitas variáveis de um tipo primitivo de dados como *int*. Cada variável cujo tipo é uma classe pode fazer referência a um único objeto dessa classe.

Nas linhas 10 a 12 do programa, as variáveis *f1*, *f2* e *f3* são preenchidas, cada uma, com um objeto do tipo *Fusca*. Variáveis cujo tipo seja uma classe não são automaticamente preenchidas com objetos quando declaradas, assim como os vetores. Para criar um novo objeto, utilizamos o comando *new*. Para utilizá-lo na criação de objetos, basta escrever o nome do comando seguido do nome da classe e após os parênteses. No programa, cada objeto criado foi atribuído a uma variável diferente.

```
1     // Programa 28: TesteFusca.java
2     // Programa que utiliza a classe Fusca.
3
4     public class TesteFusca {
5
6           public static void main(String[] args)
7           {
8                 Fusca f1, f2, f3;
9
10                f1 = new Fusca();
11                f2 = new Fusca();
12                f3 = new Fusca();
13
14                f1.cor = "Azul";
15                f1.ano = 1960;
16                f1.proprietario = "Fulano de Tal";
17
18                f2.cor = "Amarelo";
19                f2.ano = 1970;
20                f2.proprietario = "Josicreidson da Silva";
21
22                f3.cor = "Vermelho";
23                f3.ano = 1972;
24                f3.proprietario = "Florentina Souza";
25
26                System.out.println("--- Fusca 1 ---");
27                System.out.println("Cor.........: " + f1.cor);
28                System.out.println("Ano.........: " + f1.ano);
29                System.out.println("Proprietario.: " +
30                          f1.proprietario);
31
32                System.out.println("\n--- Fusca 2 ---");
33                System.out.println("Cor.........: " + f2.cor);
34                System.out.println("Ano.........: " + f2.ano);
35                System.out.println("Proprietario.: " +
36                          f2.proprietario);
37
38                System.out.println("\n--- Fusca 3 ---");
39                System.out.println("Cor.........: " + f3.cor);
40                System.out.println("Ano.........: " + f3.ano);
41                System.out.println("Proprietario.: " +
42                          f3.proprietario);
43
44          } // fim do metodo main
45
46    } // fim da classe TesteFusca
```

Programa 28: Programa que utiliza a classe Fusca.

Note que cada objeto Fusca armazena três valores diferentes. Assim como os vetores, os objetos também são *Tipos Compostos*, pois podem armazenar diversos valores. Mas ao contrário

dos vetores, os objetos são tipos *heterogêneos* de dados, pois podem possuir elementos de tipos diferentes. Cada elemento será identificado por um nome e não por um número, como nos vetores. Cada atributo dos objetos *Fusca* foi utilizado como se fosse uma variável simples.

Se as classes são tipos de dados, podemos criar vetores delas. O Programa 29 reimplementa o Programa 28 utilizando um vetor para guardar os objetos. Note que mesmo alocando o vetor na linha 8, precisamos criar um objeto para cada um dos elementos do vetor. Quando criamos um vetor de objetos, cada elemento é inicializado automaticamente com o valor *null*. Esse valor indica que aquela posição do vetor não faz referência a nenhum objeto.

```
1     // Programa 29: TesteFuscaVetor.java
2     // Programa que utiliza um vetor de objetos Fusca.
3
4     public class TesteFuscaVetor {
5
6           public static void main(String[] args)
7           {
8                 Fusca[] f = new Fusca[3];
9
10                for(int i = 0; i < f.length; i++)
11                      f[i] = new Fusca();
12
13                f[0].cor = "Azul";
14                f[0].ano = 1960;
15                f[0].proprietario = "Fulano de Tal";
16
17                f[1].cor = "Amarelo";
18                f[1].ano = 1970;
19                f[1].proprietario = "Josicreidson da Silva";
20
21                f[2].cor = "Vermelho";
22                f[2].ano = 1972;
23                f[2].proprietario = "Florentina Souza";
24
25                for(int i = 0; i < f.length; i++) {
26                      System.out.println("--- Fusca "+ (i+1) +" ---");
27                      System.out.println("Cor.........: "+f[i].cor);
28                      System.out.println("Ano.........: "+f[i].ano);
29                      System.out.println("Proprietario.: " +
30                                  f[i].proprietario + "\n");
31                }
32          } // fim do metodo main
33
34    } // fim da classe TesteFuscaVetor
```

Programa 29: Programa que utiliza um vetor de objetos.

Observe que a classe *Fusca* não foi importada nos arquivos *TesteFusca* e *TesteFuscaVetor*. Na verdade, cada classe em Java faz parte de um pacote (como as classes da API Java). Se o programador não especifica o pacote para uma classe, a classe automaticamente é colocada no pacote *default*, que inclui as classes compilados no diretório atual. Se uma classe está no mesmo

pacote que a classe que a utiliza, não é necessária uma instrução *import*. Importamos classes da API Java porque seus arquivos *.class* não estão no mesmo pacote de cada programa que escrevemos.

14.2. Definindo o Comportamento de Objetos através de Métodos

Os objetos do mundo real podem apresentar certos comportamentos quando interagimos com eles de diversas formas. Por exemplo, quando apertamos o botão *Liga/Desliga* no controle remoto de uma televisão, o aparelho pode ligar (caso esteja desligado) ou desligar (caso esteja ligado).

Os objetos que construiremos em nossos programas também apresentarão esses comportamentos. Cada comportamento será definido através de *métodos*. Já estudamos os métodos antes, mas agora iremos aplicá-los de uma forma diferente. O tipo que estudamos até agora é conhecido como *método estático*. Este método é associado diretamente a uma classe e não é necessário criar um objeto dessa classe para que ele seja utilizado. Os métodos que iremos utilizar a partir de agora, são métodos associados a objetos. Logo, só podemos utilizá-los a partir de objetos e não de classes.

Cada método irá funcionar para um objeto como um botão no controle remoto de uma televisão. Em outras palavras, um método fará alguma operação especial sobre o objeto.

Vamos analisar um exemplo. Imagine que vamos criar uma classe que represente um funcionário de uma empresa. Os dados que serão considerados para o funcionário serão nome e salário bruto, como mostrado na Figura 36.

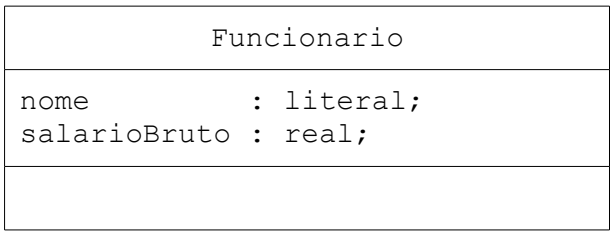


Figura 36: Atributos da classe Funcionario.

Para cada funcionário deve ser calculado o valor a ser pago ao INSS (8% do salário bruto) e o valor a ser pago ao imposto de renda (se o funcionário ganha menos de R$ 1000,00 está isento; caso contrário deve pagar 10% de seu salário). Cada um desses casos pode ser resolvido através da implementação de uma operação, ou seja de um método. A Figura 37 mostra a inclusão desses dois métodos em nosso modelo da classe.

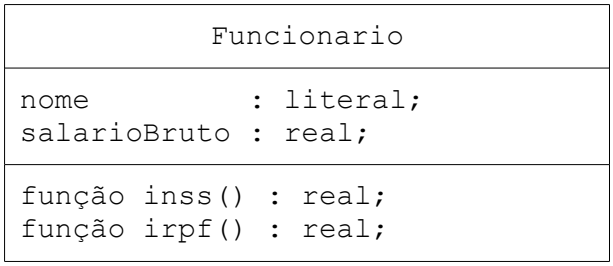


Figura 37: Modelo da classe Funcionario modificado, incluindo os métodos.

A implementação em Java da classe é mostrada na Figura 38. Veja que a definição dos métodos *inss* e *irpf* (linhas 12 a 21) difere um pouco do que estamos acostumados. Primeiro, como os dois não são métodos estáticos, a palavra *static* foi omitida. Segundo, observando o código de cada um, vemos que os atributos do objeto são diretamente acessíveis dentro dos métodos, como se fossem variáveis globais.

```
1     // Figura 38: Funcionario.java
2     // Implementacao da classe Funcionario.
3
4     public class Funcionario {
5
6           // ATRIBUTOS
7           public String nome;
8           public double salarioBruto;
9
10          // METODOS
12          public double inss() {
13                return salarioBruto * 0.08;
14          }
15
16          public double irpf() {
17                if(salarioBruto < 1000)
18                      return 0;
19                else
20                      return salarioBruto * 0.10;
21          }
22
23    } // fim da classe Funcionario
```

Figura 38: Implementação em Java da classe Funcionario.

O Programa 30 mostra um pequeno programa que faz uso da classe *Funcionario*.

```
1     // Programa 30: TesteFuncionario.java
2     // Programa que utiliza a classe Funcionario.
3     public class TesteFuncionario {
4
5           public static void main(String[] args)
6           {
7                 Funcionario func = new Funcionario();
8
9                 func.nome = "Leonardo Nascimento";
10                func.salarioBruto = 2000;
11
12                System.out.println("Nome.........: " + func.nome);
13                System.out.println("Salario Bruto.: " +
14                                        func.salarioBruto);
15                System.out.println("INSS.........: " + func.inss());
16                System.out.println("IRPF.........: " + func.irpf());
17          } // fim do metodo main
18    } // fim da classe TesteFuncionario
```

Programa 30: Programa que utiliza a classe Funcionario.

Veja que, para chamarmos um método de objeto em um programa, basta escrevermos o nome da variável que contém a referência ao objeto, um ponto final e logo após o nome do método, seguido da lista de parâmetros entre parênteses, como já vínhamos fazendo antes. Não esqueça que este tipo de método só pode ser chamado para objetos e NÃO para classes. Uma chamada `Funcionario.inss()` seria considerada um erro.

14.3. Construtores

Imagine a seguinte situação referente à classe *Funcionario* que vimos na seção anterior: toda vez que um novo objeto for criado, o salario do funcionário deve ser, por padrão, R$ 400,00. Poderíamos resolver esse problema fazendo, a cada vez que criamos um objeto do tipo *Funcionario* em nossos programas, uma atribuição para o atributo *salarioBruto* de 400. Essa solução funciona, mas é pouco elegante e complica a alteração futura desse valor padrão de salário.

Quando um objeto é criado, seus atributos podem ser inicializados por um método *construtor*. O construtor é um método com o mesmo nome da classe. O construtor é sempre chamado quando o objeto é criado, através do comando *new*. Os construtores não podem especificar tipos devolvidos nem valores devolvidos. A classe pode conter construtores sobrecarregados para oferecer diversas maneiras de inicializar os objetos dessa classe.

Como exemplo, modificaremos a classe *Funcionario* para incluir um método construtor que não recebe parâmetros. Esse tipo especial de construtor é chamado *construtor padrão*. A Figura 39 mostra a classe modificada com o novo construtor.

```
1     // Figura 39: Funcionario2.java
2     // Implementacao da classe Funcionario com construtor padrao.
3     public class Funcionario2 {
4
5           // ATRIBUTOS
6           public String nome;
7           public double salarioBruto;
8
9           // CONSTRUTORES
10          public Funcionario2() {
11                salarioBruto = 400; nome = "";
12          }
13
14          // METODOS
15          public double inss() {
16                return salarioBruto * 0.08;
17          }
18
19          public double irpf() {
20                if(salarioBruto < 1000)
21                      return 0;
22                else
23                      return salarioBruto * 0.10;
24          }
25    } // fim da classe Funcionario2
```

Figura 39: Reimplementação da classe Funcionario em Java, incluindo um construtor padrão.

A implementação do construtor é mostrada nas linhas 10 a 12. Veja que o construtor é um método como os outros, com exceção de seu nome (que deve ser exatamente igual ao da classe) e de que o tipo de retorno não é especificado.

O Programa 31 ilustra o uso da classe Funcionario2, mostrando o que acontece com o execução do construtor. A Figura 40 mostra a saída do programa.

```
1     // Programa 31: TesteFuncionario2.java
2     // Programa que utiliza a classe Funcionario2.
3     public class TesteFuncionario2 {
4
5           public static void main(String[] args)
6           {
7                 Funcionario2 func = new Funcionario2();
8
9                 System.out.println("NO INICIO DO PROGRAMA: ");
10                System.out.println("Nome.........: " + func.nome);
11                System.out.println("Salario Bruto.: " +
12                                         func.salarioBruto);
13
14                func.nome = "Leonardo Nascimento";
15                func.salarioBruto = 2000;
16
17                System.out.println("\nAPOS A ALTERACAO DO OBJETO: ");
18                System.out.println("Nome.........: " + func.nome);
19                System.out.println("Salario Bruto.: " +
20                                         func.salarioBruto);
21                System.out.println("INSS.........: " + func.inss());
22                System.out.println("IRPF.........: " + func.irpf());
23          } // fim do metodo main
24    } // fim da classe TesteFuncionario2
```

Programa 31: Programa que mostra como um construtor é executado.

```
NO INICIO DO PROGRAMA:
Nome.........:
Salario Bruto.: 400.0

APOS A ALTERACAO DO OBJETO:
Nome.........: Leonardo Nascimento
Salario Bruto.: 2000.0
INSS.........: 160.0
IRPF.........: 200.0
```

Figura 40: Saída do Programa 31.

Veja que o construtor foi executado no momento da criação do objeto *Funcionario2*, na linha 7 do programa. O construtor assegura que todo funcionário terá um salário bruto de R$ 400,00 por padrão.

Como um construtor é um método, podemos passar parâmetros a ele. A Figura 41 mostra a alteração da classe Funcionario2, para a inclusão de um construtor sobrecarregado que recebe dois parâmetros. O primeiro é o nome desejado para o funcionário e o segundo é o seu salário bruto.

```
1     // Figura 39: Funcionario3.java
2     // Implementacao da classe Funcionario2 com construtor
3     // parametrizado.
4     public class Funcionario3 {
5
6           // ATRIBUTOS
7           public String nome;
8           public double salarioBruto;
9
10          // CONSTRUTORES
11          public Funcionario3() {
12                salarioBruto = 400; nome = "";
13          }
14
15          public Funcionario3(String n, double s) {
16                salarioBruto = s; nome = n;
17          }
18
19          // METODOS
20          public double inss() {
21                return salarioBruto * 0.08;
22          }
23
24          public double irpf() {
25                if(salarioBruto < 1000)
26                      return 0;
27                else
28                      return salarioBruto * 0.10;
29          }
30    } // fim da classe Funcionario3
```

Figura 41: Classe Funcionario com um construtor parametrizado e sobrecarregado.

A classe *Funcionario3* é utilizada no Programa 32. Nas linhas 7 e 8 é apresentada a instrução que efetua a criação de um objeto da classe. Veja que, para utilizar o construtor parametrizado basta passar os valores desejados para os parâmetros, como já fazemos com os métodos.

```
1     // Programa 32: TesteFuncionario3.java
2     // Programa que utiliza a classe Funcionario3.
3     public class TesteFuncionario3 {
4
5           public static void main(String[] args)
6           {
7                 Funcionario3 func = new Funcionario3("Leonardo",
8                                                      2000);
9                 System.out.println("Nome.........: " + func.nome);
10                System.out.println("Salario Bruto.: " +
11                                   func.salarioBruto);
12                System.out.println("INSS.........: " + func.inss());
13                System.out.println("IRPF.........: " + func.irpf());
14          } // fim do metodo main
15    } // fim da classe TesteFuncionario3
```

Programa 32: Uso da classe Funcionario3 e de seu construtor sobrecarregado.

14.4. Controlando o Acesso a Atributos e Métodos

Os modificadores de acesso *public* e *private* controlam o acesso aos atributos e métodos de uma classe. Atributos e métodos definidos como *public* podem ser utilizados fora da classe onde são definidos. Todos os atributos e métodos que definimos até agora utilizaram esse tipo de acesso.

Já atributos e métodos definidos com o modificador *private* não são acessíveis pelo nome fora da classe onde estão definidos. Como exemplo, considere a classe *Aluno* definida na Figura 42. O primeiro atributo, *nome*, é do tipo *public* e pode ser alterado ou consultado diretamente fora da classe. Mas o segundo atributo, *matricula*, é do tipo *private*, e não pode ser alterado e consultado diretamente fora da classe. No Programa 33, por exemplo, há um erro na linha 9, pois o atributo *matricula* não pode ser alterado nesta classe, somente dentro da classe *Aluno*.

```
1     // Figura 42: Aluno.java
2     // Classe que possui um atributo do tipo private.
3
4     public class Aluno {
5
6           // ATRIBUTOS
7           public String nome;
8           private int matricula;
9
10    } // fim da classe Aluno
```

Figura 42: Classe com um atributo do tipo private.

Se um atributo do tipo *private* não pode ser alterado nem consultado fora da classe, qual a sua utilidade? Os desenvolvedores que utilizam OOP procuram definir todos os atributos de uma classe como *private* e alterá-los e consultá-los somente através de métodos *public* da classe. Isso permite um total controle, da parte do programador da classe, de todas as alterações e consultas que possam ser feitas no atributo. Em OOP, isso se chama *encapsulamento*, ou seja, o processo de ocultar dados dentro de um objeto.

```
1     // Programa 32: TesteAluno.java
2     // Programa que demonstra um erro ao tentar acessar um atributo
3     // private de um objeto.
4     public class TesteAluno {
5
6           public static void main(String[] args)
7           {
8                 Aluno teste = new Aluno();
9                 teste.matricula = 1234;
10                teste.nome = "Fulano";
11          } // fim do metodo main
12    } // fim da classe TesteAluno
```

Programa 33: Demonstração de um erro ao tentar alterar um atributo private de um objeto.

A Figura 43 mostra a classe *Aluno* modificada, incluindo dois métodos para acesso ao atributo *matricula*: o método *setMatricula*, para alteração do valor do atributo, e o método *getMatricula*, para consulta do valor do atributo.

```
1     // Figura 43: Aluno2.java
2     // Modificacao da classe Aluno, incluindo métodos set e get.
3
4     public class Aluno2 {
5
6           // ATRIBUTOS
7           public String nome;
8           private int matricula;
9
10          // CONSTRUTORES
11          public Aluno2() {
12                matricula = 1; nome = "";
13          }
14
15          // METODOS
16          public void setMatricula(int m) {
17                if(m > 0)
18                      matricula = m;
19          }
20
21          public int getMatricula() {
22                return matricula;
23          }
24
25    } // fim da classe Aluno2
```

Figura 43: Classe Aluno modificada, incluindo métodos set e get para alteração e consulta do atributo matricula.

Os métodos *set* são métodos criados para permitir a alteração de um atributo de um objeto.

Como regra geral de nomenclatura, costuma-se escrever a palavra *set* seguida do nome do atributo, com a primeira letra maiúscula. Os métodos *get* são usados para consulta ao valor de um atributo do objeto. A regra geral de nomenclatura segue o mesmo padrão dos métodos *get*.

Implementando um método *set* para um atributo *private*, podemos exercer um controle sobre o valor do atributo que não exerceríamos caso o atributo fosse *public*. No exemplo, não permitimos que o atributo *matricula* seja alterado para um valor não-positivo, pois seriam valores inválidos para um número de matrícula. Se o atributo fosse definido como *public*, poderia ser alterado para qualquer valor aceito pelo tipo *int*, incluindo o zero e números negativos. O Programa 34 mostra o Programa 33 modificado, solucionando o problema anterior através da chamada do método *setMatricula*.

```
1     // Programa 34: TesteAluno2.java
2     // Programa que demonstra o uso de um metodo set.
3
4     public class TesteAluno2 {
5
6           public static void main(String[] args)
7           {
8                 Aluno teste = new Aluno();
9                 teste.setMatricula(1234);
10                teste.nome = "Fulano";
11
12                System.out.println("Nome......: " + teste.nome);
13                System.out.println("Matricula.: " +
14                            teste.getMatricula());
15          } // fim do metodo main
16    } // fim da classe TesteAluno2
```

Programa 34: Exemplo de uso de métodos set e get.

14.5. Exercícios

1. Defina e implemente em Java classes e atributos (sem definir métodos) para representar os objetos abaixo. Ao lado de cada item é mostrado um exemplo com os dados que devem ser armazenados. Escreva classes com atributos públicos.

a) Pessoa Física

```
Nome: José da Silva
CPF: 123.456.789-00
RG: 1234567890
Idade: 30
Sexo: M // 'M' para masculino e 'F' para feminino
```

b) Sapato em uma loja

```
Descrição: Tênis Kichute
Preço: 20,00
Tamanho: 40
Cor: P // 'P' p/ preto, 'A' p/ azul e 'B' p/
branco
```

c) Livro em uma biblioteca

Título: Java, como Programar
Autor: Deitel & Deitel
Emprestado: true
Nro de Paginas: 1000
Edição: 4

d) Item em uma nota fiscal

Descrição: Mouse Óptico
Preço Unitário: 25,00
Quantidade: 4

e) Histórico escolar de um aluno do curso de Informática

Nome: Fulano de Tal
Matricula: 1234
Ano de Ingresso: 2005
Semestre de Ingresso: 2

Módulo	***Conceito***
Lógica de Programação	A
Bancos de Dados	B
Engenharia de Software	A

2. Escreva um programa em Java que realize uma pesquisa em um banco de dados de pessoas. O programa deve ler um CPF de uma pessoa. Caso a pessoa esteja cadastrada no banco de dados, o programa deve mostrar suas informações na tela. Caso contrário, deve mostrar a mensagem **"Pessoa não cadastrada"**. Para simular o banco de dados, crie um vetor de 10 posições. O tipo dos elementos do vetor será PessoaFisica (a classe que você definiu no exercício 1 item a). Preencha o vetor com dados de pessoas de sua escolha. Quando o usuário digitar um CPF, o programa deverá comparar o CPF digitado com o de cada pessoa presente no vetor. Caso encontre um CPF igual, então a pessoa foi encontrada; caso contrário, a pessoa não está cadastrada.

3. Escreva um programa que, a semelhança do exercício 2, crie um vetor de 15 objetos do tipo LivroBiblioteca (classe criada no exercício 1 item c). Preencha o vetor com os valores que você desejar. Escreva um programa que mostre na tela os livros que estão disponíveis na biblioteca, ou seja, aqueles que não estão emprestados.

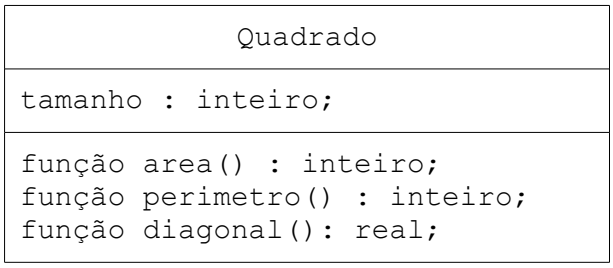
4. Reescreva a classe desenvolvida no exercício 1 item d, incluindo um método que calcule o valor total do item.

5. Escreva um programa que utilize a classe definida no exercício 4 para gerar uma nota fiscal completa. O programa deve ler o número de itens que nota deverá conter e os dados de cada item (descrição, preço unitário e quantidade). Na nota deverão aparecer os dados de cada item, o valor total de cada um, o total da nota (soma dos valores totais de todos os itens) e o ICMS a ser pago (12% do valor total da nota).

6. Reescreva a classe sapato do exercício 1, item b, encapsulando seus atributos (colocando todos eles como *private*). Crie métodos *set* e *get* apropriados. Não permita que a cor possa ser alterado para valores diferentes de 'P', 'A' ou 'B'.

7. Escreva um programa que use a classe definida no exercício 6 para realizar consultas em um banco de dados de sapatos. As consultas podem ser feitas de duas formas: por preço ou por tamanho. A consulta por preço permite que o usuário veja todos os sapatos que custam menos do que o preço informado por ele. A consulta por tamanho permite que o usuário visualize todos os sapatos que possuem um tamanho digitado pelo usuário. O banco de dados deve ser criado de forma similar ao dos exercícios 2 e 3.

8. Crie uma classe Quadrado, de acordo com o modelo mostrado abaixo. Os atributos da classe devem ser encapsulados. Crie os métodos *set* e *get* apropriados. Inclua um construtor na classe que receba um parâmetro do tipo *int* e altere o valor do atributo *tamanho*, a partir desse valor.



9. Escreva um programa em Java que leia o tamanho de vários quadrados, até que o usuário digite um valor negativo para o tamanho. Utilize um objeto do tipo Quadrado para armazenar o valor do tamanho e calcular a área, o perímetro e a diagonal do quadrado, e mostrar os resultados na tela.

10. Refaça os exercícios 2, 3, 4 e 5 utilizando classes com atributos encapsulados. Crie os métodos *set* e *get* necessários. Faça, nos métodos *set*, a validação necessária para cada atributo, se necessário. Crie um construtor sem parâmetros para cada classe, que inicialize os atributos com valores padrão definidos por você. Todos os atributos devem ser inicializados nos construtores.