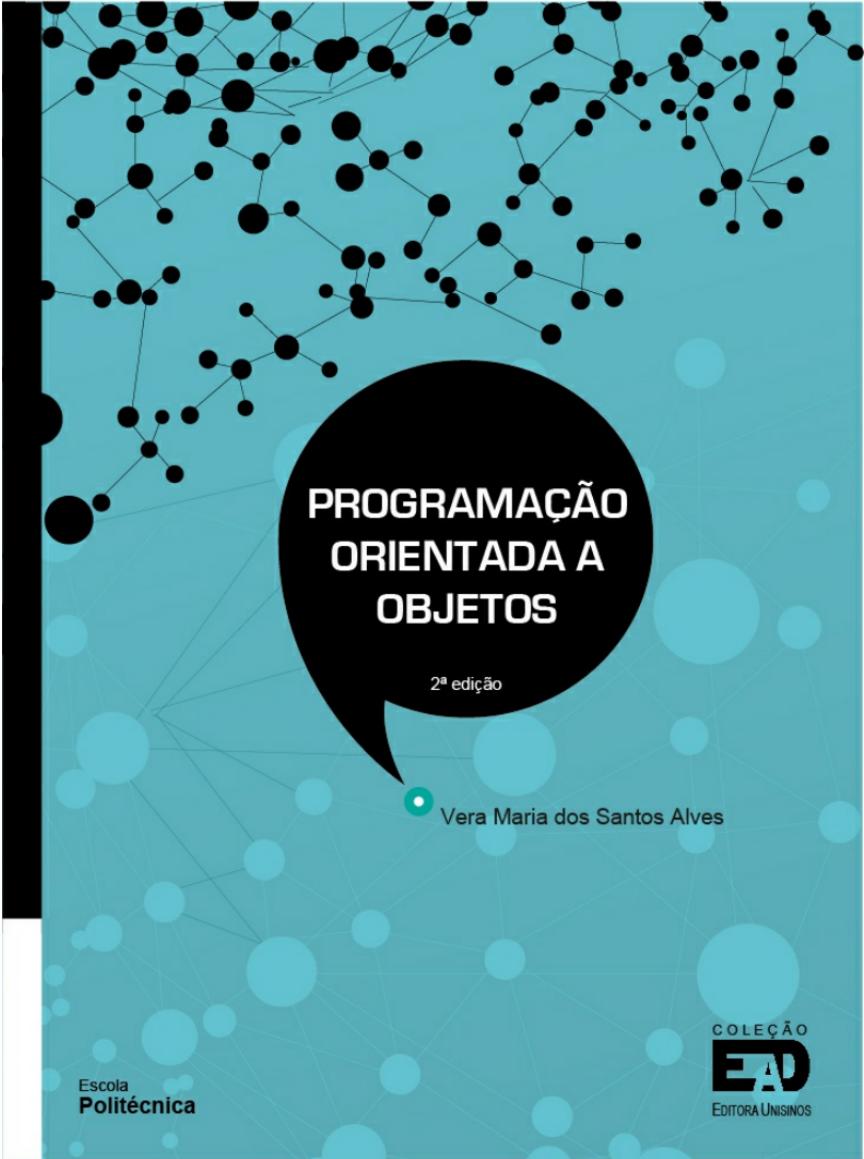


PROGRAMAÇÃO ORIENTADA A OBJETOS

2^a edição



Vera Maria dos Santos Alves



PROGRAMAÇÃO ORIENTADA A OBJETOS

2ª edição

Vera Maria dos Santos Alves

Escola
Politécnica

COLEÇÃO
EAD
EDITORA UNISINOS

PROGRAMAÇÃO ORIENTADA A OBJETOS

VERAMARIA DOS SANTOS ALVES

2^a edição

Editora Unisinos, 2014

SUMÁRIO

[Apresentação](#)

[Capítulo 1 – Introdução à programação](#)

[Capítulo 2 – Elementos básicos](#)

[Capítulo 3 – Os primeiros comandos](#)

[Capítulo 4 – Classes e objetos](#)

[Capítulo 5 – Métodos de configuração e acesso aos atributos](#)

[Capítulo 6 – Classe de teste](#)

[Capítulo 7 – Abiblioteca Java](#)

[Capítulo 8 – Estruturas de seleção](#)

[Capítulo 9 – Estruturas de repetição](#)

[Capítulo 10 – Herança e polimorfismo](#)

[Capítulo 11 – Arrays](#)

[Capítulo 12 – Exercícios resolvidos](#)

[Sobre a autora](#)

[Informações técnicas](#)

APRESENTAÇÃO

Este material foi elaborado para conduzir o leitor desde o entendimento dos conceitos básicos da programação de computadores até o desenvolvimento de programas simples, orientados a objetos, na linguagem Java.

Levando em conta a minha experiência no ensino de programação, desde o tempo dos cartões perfurados, e tendo acompanhado toda a evolução da computação, posso dizer a quem está ingressando nessa área que o conhecimento teórico é o que dá sustentação ao profissional para enfrentar as constantes mudanças e os novos desafios.

Ao longo dos anos foram surgindo diferentes metodologias para desenvolver software e, consequentemente, a programação de computadores, que é uma das fases do desenvolvimento de software, tem acompanhado essas mudanças de paradigmas. Hoje, programamos seguindo os princípios do paradigma “orientado a objetos”, independentemente da linguagem.

Ao escrever um programa de computador em qualquer linguagem de programação, você está abstraindo alguma parte da realidade, em um contexto particular. O contexto é o que chamamos de domínio da aplicação ou negócio. Abstrair significa captar as informações importantes, que fazem sentido à aplicação particular, e criar um modelo que represente a realidade e que possa ser programado. No paradigma orientado a objeto, as entidades identificadas na realidade são representadas por objetos. Um conjunto de objetos, com características semelhantes são descritos por uma classe. Os dois termos, *classe* e *objeto*, provavelmente ainda com significados nebulosos, a partir de agora devem entrar não só para o seu vocabulário, como também para a sua mente, de forma que, à medida que você for lendo este material, a visualização desses conceitos vá ficando cada vez mais nítida, mais rica em detalhes e, surpreendentemente, cada vez mais simples.

Voltando a enfatizar a importância da teoria, recomendo a leitura cuidadosa e o entendimento de cada conceito, e para isso se faz necessária a sua aplicação em um programa. Aí entra a parte prática que deve complementar o aprendizado, pois a melhor forma de aprender a programar é programando.

Por fim, um esclarecimento quanto à linguagem adotada. A ideia deste livro não é ensinar Java, mas usá-la como ferramenta para praticar a programação e para garantir que o leitor domine os conceitos fundamentais que lhe permitam migrar para qualquer outro ambiente,

conforme suas necessidades atuais e futuras.

Aautora

CAPÍTULO 1

INTRODUÇÃO À PROGRAMAÇÃO

Este capítulo visa capacitar o leitor a acompanhar os próximos capítulos e dar início à aprendizagem de programação de computadores. Aqui serão apresentados alguns conceitos fundamentais, necessários a alguém que está aprendendo a programar.

1.1 O que é um computador

Um computador é uma máquina constituída por uma combinação de componentes eletrônicos e que pode ser programada. A parte física é denominada hardware. Para que essa máquina seja útil, ela deve ser dotada de software, que são os programas. Isso significa que o computador sempre está sob o controle de um programa, ou seja, está fazendo aquilo que o programa determina. Dizemos que o computador está rodando ou executando o programa. Então, o que faz um computador ser uma máquina fantástica e eclética é a alta velocidade e a possibilidade de ser programado, isso é, o software dita o comportamento do hardware.

Software é um conjunto de programas e um programa é um conjunto de comandos ou instruções que devem ser realizados para executar uma tarefa.

Rodar ou executar um programa significa que o computador irá realizar cada um dos comandos ou instruções, na sequência especificada.

O computador, de modo geral, é formado por cinco blocos principais, ou seja, cinco unidades para as quais o programa determinará as tarefas a serem executadas.

1.2 Unidades básicas de um computador

A arquitetura básica dos modernos computadores ainda é baseada na arquitetura proposta por Von Neumann há aproximadamente meio século, que tinha como principal característica o armazenamento de programas no mesmo espaço de memória que os dados.

Atualmente, para fins didáticos, podemos dividir os computadores em unidades lógicas ou seções, conforme a figura a seguir.

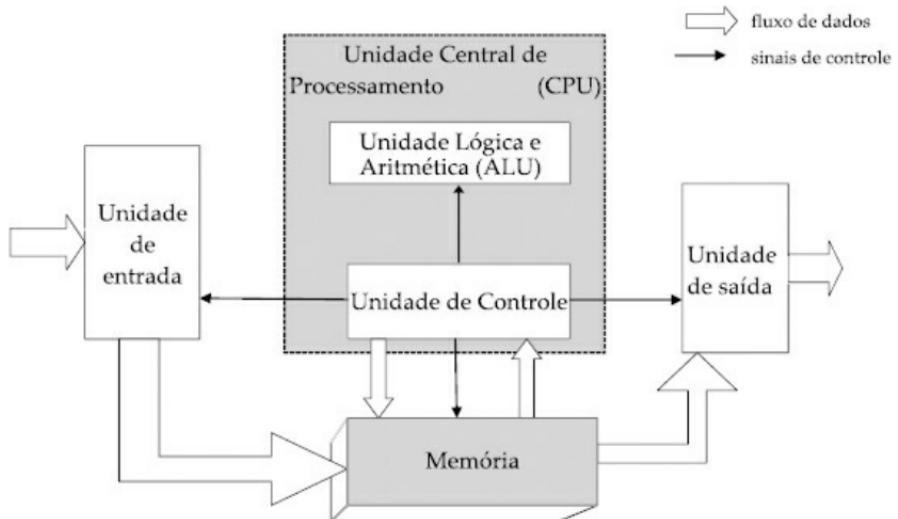


Figura 1 – As unidades básicas de um computador.
Fonte: domínio público.

1.2.1 Descrição das unidades básicas

A seguir, uma breve descrição das unidades básicas do computador:

- *Unidade Central de Processamento* – UCP (em inglês, *Central Processing Unit* – CPU). É a responsável pelo processamento e pela execução dos programas armazenados na memória principal. As funções da CPU são: executar as instruções e controlar as operações no computador. A CPU possui, além de registradores, duas unidades importantes: a Unidade de Controle e a Unidade Lógica e Aritmética.
- *Unidade Lógica e Aritmética* – ULA (em inglês, *Arithmetic Logic Unit* – ALU). É onde ocorre a execução das operações aritméticas e lógicas. O tipo de operação a ser executada é determinado pela unidade de controle, que envia os sinais adequados.
- *Unidade de Controle* – UC. Tem por funções a busca, a interpretação e o controle de execução das instruções, além do controle dos demais componentes do computador.

- *Unidades de Entrada/Saída*. Chamadas de dispositivos de E/S. Permitem a troca de informações com o exterior. Um exemplo de um dispositivo de entrada é o teclado. Um exemplo de um dispositivo de saída é o monitor, que exibe números, textos e imagens em uma tela de vídeo.
- *Memória principal*. Também conhecida como memória RAM. Armazena, temporariamente, programas e dados.

Em sistemas digitais, os dados armazenados e processados estão na forma binária, ou seja, são representados por uma combinação de apenas dois símbolos ou valores possíveis: 0 e 1, que são conhecidos como dígitos binários ou bits.

Assim, a unidade mínima de informação é o bit (*BInarydigiT*). Um agrupamento de oito bits é denominado byte. A capacidade de memória é dada em bytes, conforme especificado abaixo:

- 1 byte = 8 bits
- 1 KB (kilobyte ou kbyte) = $2^{10} = 1024$ bytes
- 1 MB (megabyte) = $2^{20} = 1 \text{ KB} * 1 \text{ KB} = 1.048.576$ bytes
- 1 GB (gigabyte) = $2^{30} = 1 \text{ KB} * 1 \text{ MB} = 1.073.741.824$ bytes
- 1 TB (terabyte) = $2^{40} = 1 \text{ KB} * 1 \text{ GB} = 1.099.511.627.776$ bytes

1.3 Como funciona o computador

O computador funciona sob o controle de um programa armazenado, executando instrução por instrução. Tanto a sequência de instruções como os dados estão na forma binária. Diz-se que o programa está em linguagem de máquina, que é a única linguagem entendida pelo computador.

O leigo ou um usuário comum (não especialista) normalmente tem uma visão não realista de um computador, pois tende a ver um computador não apenas como uma máquina e sim como uma soma de software e hardware, sem a preocupação de distinguir o que é o software e o que é o hardware. Quando um usuário liga um computador, o seu contato é com o software que está instalado no computador, que é o sistema operacional (pode ser Windows, Linux etc.). Esse software permite então que outros programas sejam executados.

O programador deve enxergar o computador como uma máquina

passível de ser programada e deve ter em mente que o comportamento do computador depende do software que ele está executando no momento.

1.4 Desenvolvimento de um software

A programação é uma das últimas fases do desenvolvimento de um software, pois antes da programação deve-se analisar e projetar a solução.

O software deve surgir como resposta às necessidades do mundo real e deve atender adequadamente a essas necessidades.

Normalmente, essas necessidades são especificadas por um usuário, que é quem entende do negócio. Cabe ao projetista do software abstrair as características relevantes e construir um modelo que represente o comportamento do software. Esse processo denominamos modelagem, e podemos utilizar uma linguagem própria para essa fase, independente da linguagem de programação. A modelagem deve servir tanto para a comunicação com o usuário como para o planejamento do software que será construído.

Ao longo do tempo, à medida que as necessidades e expectativas dos usuários foram aumentando em quantidade e complexidade, novos processos de desenvolvimento de software foram sendo necessários, e metodologias foram sendo criadas para ajudar a construir esse produto. A essas diferentes formas de abordagem damos o nome de “paradigmas”.

Considerando que paradigma é a forma de organizar e visualizar o conjunto de programas que constitui o software, então o paradigma adotado é que determina a forma como abstraímos e, consequentemente, desenvolvemos o software.

O paradigma orientado a objetos pode ser utilizado na análise, no projeto e na programação do software. Aqui estamos interessados na última fase do desenvolvimento de um software, ou seja, a programação, onde os modelos criados são convertidos em código. Utilizaremos a Programação Orientada a Objetos (POO), ou ainda, em inglês, *Object-Oriented Programming* (OOP), que é um paradigma baseado na modularização em unidades de software, chamadas classes. A partir das classes são criados objetos que, em tempo de execução, interagem e cooperam entre si.

1.5 O que é programação de computador

Programar para um computador consiste em desenvolver um software que possa ser executado no computador.

Para escrever um programa precisamos de uma linguagem de programação. A única linguagem entendida pelo computador é a linguagem de máquina, que é uma linguagem difícil para o ser humano, pois tudo é especificado usando apenas os dígitos binários (0s e 1s).

Devido a essa dificuldade, foram criadas as chamadas linguagens de alto nível, que são facilmente entendidas pelas pessoas. O programador escreve o programa em uma linguagem de alto nível (programa-fonte ou código-fonte) e o programa é traduzido para a linguagem de máquina (programa-objeto ou código-objeto) para que o computador possa executá-lo.

A tradução é realizada por um software denominado compilador, que gera instruções correspondentes, na linguagem de máquina, de um processador específico. É importante notar que o compilador somente gera o código-objeto caso não tenha sido encontrado nenhum erro sintático no programa-fonte, ou seja, não tenha sido violada nenhuma regra da linguagem de programação utilizada.

Cada linguagem de programação de alto nível deve ter o seu compilador para uma determinada arquitetura, já que o código executável produzido não é portável: diferentes compiladores/montadores são construídos para as diferentes arquiteturas de processadores, pois diferentes famílias de processadores possuem conjuntos distintos de instruções.

1.6 Como é programar em Java

O compilador da linguagem Java não gera código para um processador específico, mas sim para uma máquina virtual Java (conhecida como *Java Virtual Machine – JVM*). Os programas escritos em Java são, portanto, portáveis sobre todas as plataformas para as quais existir uma implementação da JVM. Programas Java são constituídos de classes. Os membros de uma classe são os atributos (ou campos) e os métodos. O programador pode, além de escrever suas próprias classes, usar as classes prontas da biblioteca Java, conhecida como Interface do Programa de Aplicação (API).

O programa-fonte é criado em um editor e armazenado em um arquivo com a extensão *.java*. O compilador lê o arquivo *.java* e traduz o programa fonte em bytecodes e armazena em um arquivo com a extensão

.class. O carregador lê o arquivo .class e carrega os bytecodes na memória. O verificador de bytecodes confirma a validade dos bytecodes quanto às restrições de segurança. Finalmente, a máquina virtual executa o programa, seguindo as ações especificadas pelos bytecodes, que nada mais são do que os comandos codificados pelo programador, porém na linguagem da máquina.

O passo a passo que deve ser seguido pelo programador iniciante tem início em um editor de texto (bloco de notas ou um editor específico para Java), onde ele começa digitando o seu programa. Após a digitação, o programa é submetido ao compilador tantas vezes quantas forem necessárias, pois o compilador tenta traduzir o programa escrito em Java para bytecode e, cada vez que encontra um erro, exibe uma mensagem indicando o tipo do erro. A cada erro, o programador deve voltar ao editor, arrumar o erro e novamente submeter o programa ao compilador. Quando o programa estiver sintaticamente correto, o compilador realiza a tradução e o programa pode ser executado pela JVM. A execução do programa não significa que o programa esteja “funcionando”, significa apenas que não há erros de sintaxe. Somente podemos dizer que um programa está “funcionando” após efetuarmos vários testes e observarmos que o comportamento do programa é o que esperávamos.

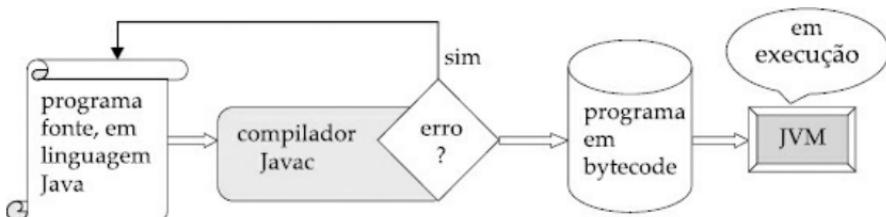


Figura 2 – As fases: Edição, compilação e execução.
Fonte: elaborada pela autora.

1.7 Erros

Durante o processo descrito acima podemos identificar três fases: edição, compilação e execução, que podem ser repetidas ou interrompidas devido à ocorrência de erros. Dependendo da fase em que o erro ocorre, podemos categorizar os tipos de erro abaixo:

- *Erro de compilação ou de sintaxe* – É um erro que ocorre quando alguma regra da linguagem de programação é violada. Esse erro é detectado pelo compilador, que interrompe o processo de tradução e notifica o programador através de uma mensagem. Nesse caso, o arquivo contendo o bytecode não é gerado. O programador deve voltar à fase de edição.
- *Erro de execução* – É um erro que ocorre durante a execução do programa, que é interrompida. Esse erro é detectado pela JVM, que acusa o erro lançando uma exceção (exception) e notificando o programador com uma mensagem indicando a exceção. Um exemplo de exceção seria uma divisão por zero. O programador deve analisar a situação e voltar à fase de edição.
- *Erro de lógica* – Esse erro também é conhecido como *bug*. O programa foi corretamente compilado e executado, porém o resultado obtido não é o desejado. Detectado pelo programador ou pelo testador ou até mesmo pelo usuário final.

1.8 Comandos

Comandos ou instruções são as ordens dadas ao computador para que este realize as tarefas especificadas pelo programador.

Um conjunto de comandos, em uma determinada sequência, pode ser denominado de algoritmo ou programa, dependendo da linguagem utilizada. Em geral, a linguagem algorítmica é uma linguagem mais próxima da língua falada e serve para especificar a lógica de um problema de forma clara, sem a preocupação com detalhes técnicos e rígidos de uma linguagem de programação. Outra grande vantagem da linguagem algorítmica em relação a uma linguagem de programação é que qualquer programador consegue entender um algoritmo. A desvantagem é que o algoritmo não pode ser executado pelo computador a não ser que seja construído um tradutor para essa linguagem, mas, neste caso, esta também passa a ficar submetida a um conjunto de regras rígidas, perdendo a flexibilidade, restando apenas a clareza.

Ao programar, independentemente da linguagem, devemos saber dois aspectos a respeito dos comandos: a sintaxe e a semântica.

A sintaxe diz respeito à forma do comando, ou seja, qual a palavra-chave e quais os elementos que dele fazem parte. A maioria dos comandos tem uma palavra-chave que os definem. Exemplo de palavras-chave que identificam os comandos: *if* (em português, *se*), *while* (em

português, *enquanto*), *do* (em português, *faça*), *for* (em português, *para*).

A semântica refere-se ao significado do comando, ou seja, o que o comando faz.

O programador necessita estar ciente de que tudo o que o computador deve fazer precisa ser detalhadamente especificado no programa, através dos comandos. O programa deve dizer o que fazer e em que ordem as instruções devem ser realizadas.

O raciocínio lógico usado pelo programador diz respeito a saber exatamente o que acontece nas unidades do computador quando o comando é executado. Considerando que um comando é uma ordem que será executada pelo computador, o programador, que é quem emitiu a ordem, deve conhecer as consequências, ou seja, qual é o resultado da ação.

Os comandos podem ser classificados em duas categorias: os que atuam nas três unidades básicas do computador (memória, entrada e saída) e os que controlam o fluxo de execução do programa.

Na primeira categoria temos os comandos que atuam diretamente nas unidades do computador, armazenando valores na memória, fazendo cálculos, lendo valores da entrada e exibindo resultados na tela. Aqui temos os comandos de entrada e saída e o comando de atribuição.

Os comandos de entrada e saída são as instruções que devem ser dadas para que o computador se comunique com o meio exterior. Estes comandos atuam nas unidades de entrada e de saída do computador, envolvendo a unidade de memória para armazenar valores oriundos da unidade de entrada e para obter valores para serem exibidos na unidade de saída.

Um comando de entrada, também conhecido como comando de leitura, faz com que o computador capture um valor de uma unidade de entrada. Por exemplo, se a unidade de entrada for o teclado, quando este comando é atingido o computador aguarda até que um valor seja digitado no teclado e então o armazena na memória.

Um comando de saída, também conhecido como comando de escrita, faz com que o computador escreva na unidade de saída. Por exemplo, se a unidade de saída for a tela, quando este comando é atingido o computador exibe na tela os itens especificados no comando.

Um comando de atribuição atua na unidade de memória do computador. A forma como o programador se refere à memória é através das variáveis. O comando de atribuição armazena valores nas variáveis. Esses valores podem ser constantes, valores de outras variáveis ou resultados de cálculos.

Na segunda categoria estão os comandos de controle, que dizem

respeito ao fluxo de execução, pois permitem que um programa seja executado de forma não linear, realizando desvios e repetições, conforme o resultado de condições avaliadas durante a execução. Nesta categoria temos os comandos de seleção e os comandos de repetição.

Os comandos de seleção são responsáveis pelas “decisões”, baseadas em condições que o computador deve avaliar durante a execução do programa. Os comandos de repetição ou iteração são os que permitem ao programador especificar a repetição de certos trechos do programa.

CAPÍTULO 2

ELEMENTOS BÁSICOS

Neste capítulo daremos início ao aprendizado de uma linguagem de programação de alto nível, apresentando os elementos básicos que fazem parte da maioria das linguagens de programação. Para que esse aprendizado possa ser visto na prática, é necessário adotarmos uma linguagem em particular, então, veremos os elementos básicos da linguagem Java.

2.1 Dados

Os dados são armazenados e processados no computador. O programador determina o armazenamento e a manipulação dos dados através de instruções próprias para esse fim. Na prática, costumamos usar os termos *dado* e *informação* como sinônimos, embora seja importante fazer a distinção.

2.1.1 Dado

Dado é a representação de algum elemento da realidade. Essa representação é feita através de letras, dígitos e caracteres especiais. Internamente, esses dados são armazenados como uma combinação de bits. O computador manipula e *enxerga* dados, ou seja, conjunto de bits sem nenhum significado.

Exemplos de dados:

- 35924670
- M
- Maria dos Santos
- XW34
- 1947,23
- falso

2.1.2 Informação

Informação é o significado de um dado. O ser humano (*programador, usuário*) trabalha com informações. Os dados exemplificados acima, que são armazenados no computador como uma combinação de bits, representam informações para as pessoas.

Exemplos de informações:

- 35924670 pode ser o telefone da empresa X.
- Maria dos Santos pode ser o nome da diretora de empresa X.
- XW34 pode ser o código de uma peça fornecida pela empresa X.
- 1947,23 pode ser o salário-base oferecido na empresa X.

2.1.3 Estrutura de dados

Estrutura de dados é um conjunto de dados relacionados. Em ciência da computação estudamos diferentes formas de se organizar os dados para armazenamento e posterior recuperação. Cada forma de organizar e recuperar os dados é conhecida como uma estrutura de dados ou uma coleção.

Exemplos de estruturas de dados:

- uma estrutura constituída pelos funcionários da empresa X, em que cada elemento dessa estrutura armazena os dados de um determinado funcionário;
- uma coleção constituída pelas peças fornecidas pela empresa X, em que cada elemento dessa coleção armazena os dados de uma determinada peça.

2.1.4 Tipos de dados

Os dados representam as informações da vida real, então podemos ter dados que usam apenas valores inteiros (sem parte fracionária), outros que possuem uma parte fracionária após a vírgula ou após o ponto decimal e são conhecidos como valores reais ou de ponto flutuante, outros que usam caracteres e são conhecidos como literais e, ainda, os lógicos ou booleanos, que representam os valores falso e verdadeiro.

| Dado: | Tipo: |
|------------------|-----------------------------------|
| 35924670 | número inteiro |
| M | caractere ou literal |
| Maria dos Santos | sequência de caracteres |
| XW34 | sequência de caracteres |
| 1947,23 | número real ou de ponto flutuante |
| falso | lógico ou booleano |

As linguagens de programação oferecem uma gama de tipos de dados, contemplando os tipos descritos acima e diferentes tamanhos de espaços de armazenamento na memória, associando cada tipo com um identificador próprio da linguagem (nome do tipo). Os tipos de dados oferecidos por uma linguagem de programação são denominados tipos simples ou primitivos. Abaixo, os tipos primitivos da linguagem Java, com o espaço em bytes, ocupado na memória, e o intervalo de valores suportado por cada tipo.

Quadro 1 – Tipos de dados primitivos da linguagem Java

| Identificador do Tipo | Espaço de armazenamento | Intervalo de valores |
|------------------------------|--------------------------------|--|
| int | 4 bytes | -2.147.483.648 a 2.147.483.647 |
| short | 2 bytes | -32.768 a 32.767 |
| long | 8 bytes | -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 |
| byte | 1 byte | -128 a 127 |
| float | 4 bytes | $\pm 3.402823347E+38$ (7 dígitos significativos) |
| double | 8 bytes | $\pm 1.79769313486231570E+308$ (15 dígitos significativos) |
| char | 2 bytes | qualquer caractere Unicode |
| boolean | 1 byte | false, true |

Fonte: Horstmann, Cay S. Core Java.

2.2 Constantes

Constantes são valores imutáveis que expressam os valores dos dados. As constantes podem ser de qualquer tipo.

Exemplos de constantes:

- 35924670 é uma constante do tipo int
- M é uma constante do tipo char
- 1947.23 é uma constante do tipo double
- false é uma constante do tipo *boolean*

2.2.1 Como as constantes numéricas são interpretadas em Java

Os números escritos sem ponto decimal são interpretados como do tipo int. Se o valor extrapolar os limites do tipo int, será tratado como inteiro muito grande e dará erro de compilação. Para que um inteiro seja tratado como do tipo long, devemos colocar o sufixo L ou l junto ao número. Os números escritos com ponto decimal são interpretados como do tipo double. Para que um número seja tratado como do tipo float devemos colocar o sufixo F ou f junto ao número. O sufixo D ou d também pode ser usado para informar ao compilador que desejamos tratar o número como double.

Exemplos de constantes numéricas com e sem sufixo:

- 1253722 é tratada como int
- 1253722 L é tratada como long, devido ao sufixo
- 2999999999 é tratada como int e dá o erro de compilação “*integer number too large*”
- 3.1415 é tratada como double
- 3.1415F é tratada como float devido ao sufixo
- 15D é tratada como double, devido ao sufixo

2.2.2 Constantes com identificadores

As constantes podem ser declaradas usando a palavra reservada *final* e são também conhecidas como variáveis finais. Então, uma declaração de constante consiste em colocar a palavra reservada *final*, o tipo e o identificador (nome) da constante. A inicialização pode ocorrer junto com a declaração. Depois de inicializada, não pode mais ser alterada, por isso a denominação de constante.

Convencionalmente, os identificadores de constantes devem ser escritos em letras maiúsculas, usando underline (_) para separar nomes compostos.

Exemplos de declarações, com inicializações de constantes (variáveis *final*):

- `final double TAXA_DE_DESCONTO = 0.27` especifica que `TAXA_DE_DESCONTO` é uma constante de valor 0.27
- `final int TAMANHO = 10` especifica que `TAMANHO` é uma constante de valor 10.

2.3 Variáveis

Variáveis são nomes simbólicos de endereços de memória, em que os dados ou as referências são armazenadas. As variáveis podem mudar de valor durante a execução do programa. Os identificadores (nomes) das variáveis são livremente escolhidos pelo programador, respeitando as regras da linguagem.

Em Java, trabalhamos com duas categorias de variáveis: as variáveis do tipo primitivo e as variáveis-referência.

As variáveis do tipo primitivo são as que armazenam dados do tipo primitivo, tais como números, caracteres e valores lógicos.

As variáveis-referência não armazenam dados, e sim referências (endereços de memória) dos dados não primitivos, ou seja, dos objetos, que serão apresentados no Capítulo 4. Podemos dizer que a variável do tipo primitivo armazena o próprio dado e a variável-referência aponta para o dado.

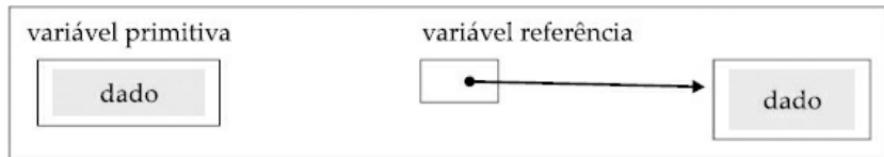


Figura 3 – As variáveis, em Java.

Fonte: elaborada pela autora.

Exemplos de variáveis:

- salario
- a0
- anoAniver
- bloqueada
- nome
- saldo
- n1
- s
- salLiquido

2.3.1 Identificadores das variáveis

Os identificadores (nomes) das variáveis devem obedecer às regras da linguagem e devem ser escolhidos pelo programador levando em conta a finalidade da variável, ou seja, o que a variável armazena. Por exemplo, uma variável para armazenar um salário pode ter o nome de *sal* ou *salário*, que identifica o dado que será armazenado na variável, mas nada impede que o nome da variável seja *x*, que não tem nenhum significado. Em Java, os identificadores de variáveis podem conter letras, dígitos e os símbolos _ e \$, mas não podem ter um dígito como primeiro caractere. As palavras reservadas da linguagem Java não podem ser utilizadas como identificadores.

2.3.2 Declaração das variáveis

As variáveis devem ser declaradas no programa, antes de serem

utilizadas. A declaração serve para que o compilador aloque um espaço de memória adequado ao tipo da variável. Então, uma declaração de variável consiste em especificar o tipo e o nome da variável.

Exemplos de declarações de variáveis primitivas:

- `double salario` especifica que `salário` é o nome de uma variável que pode armazenar um número real.
- `int n1` especifica que `n1` é o nome de uma variável que pode armazenar um número inteiro.
- `int x, y` especifica que `x` é o nome de uma variável que pode armazenar um número inteiro e que `y` é o nome de outra variável que também pode armazenar um número inteiro.
- `double notaGa` especifica que `notaGa` é o nome de uma variável que pode armazenar um número real.
- `boolean bloqueada` especifica que `bloqueada` é o nome de uma variável que pode armazenar um valor booleano (`false` ou `true`).

2.4 Expressões

Para especificar as operações que devem ser realizadas sobre os dados, a linguagem permite o uso de expressões, que podem usar operadores aritméticos, relacionais e lógicos sobre variáveis e constantes. Os resultados podem ser atribuídos a variáveis, através dos operadores de atribuição, em um comando de atribuição. As expressões aritméticas dão como resultado um número (do tipo inteiro ou real). As expressões relacionais e lógicas dão como resultado um valor lógico (`false` ou `true`).

2.4.1

Quadro 2 – Operadores aritméticos

| operador | descrição | exemplo | resultado |
|----------------|---------------|---------|-----------|
| <code>+</code> | soma | $5+3$ | 8 |
| <code>-</code> | subtração | $5-3$ | 2 |
| <code>*</code> | multiplicação | $5*3$ | 15 |

| | | | |
|---|------------------|------|---|
| / | divisão | 12/3 | 4 |
| % | resto da divisão | 12%3 | 0 |

Fonte: elaborado pela autora.

2.4.2

Quadro 3 – Operadores relacionais

| operador | descrição | exemplo | resultado |
|----------|------------------|--------------|-----------|
| > | maior que | 5>3 | true |
| < | menor que | 5<3 | false |
| \geq | maior ou igual a | 5 \geq 3 | true |
| \leq | menor ou igual a | 5 \leq 3 | false |
| \equiv | igual a | 5 \equiv 3 | false |
| \neq | diferente de | 5 \neq 3 | true |

Fonte: elaborado pela autora.

2.4.3

Quadro 4 – Operadores lógicos

| operador | descrição | exemplo | resultado |
|----------|---------------------|----------------|-----------|
| ! | negação (NOT) | !(5>3) | false |
| $\&\&$ | e condicional (AND) | 5>3 $\&\&$ 2>4 | false |
| $\ $ | ou condicional (OR) | 5>3 $\ $ 2>4 | true |

Fonte: elaborado pela autora.

2.4.4

Quadro 5 – Operadores de atribuição

| operador | descrição | exemplo | resultado na memória |
|----------|--------------------------------|---------|---|
| = | atribuição | s=1 | a variável s assume o valor 1 |
| += | atribuição de soma | s +=3 | a variável s assume o valor 4 (é o mesmo que s=s+3) |
| -= | atribuição de subtração | s -=1 | a variável s assume o valor 3 (é o mesmo que s=s-1) |
| *= | atribuição de multiplicação | s *=4 | a variável s assume o valor 12 (é o mesmo que s=s*4) |
| /= | atribuição de divisão | s /=2 | a variável s assume o valor 6 (é o mesmo que s=s/2) |
| %= | atribuição de resto da divisão | s %=3 | a variável s assume o valor 0 (é o mesmo que s=s%3) |

Fonte: elaborado pela autora.

2.4.5

Quadro 6 – Operadores unários

| operador | descrição | exemplo | resultado na memória |
|----------|------------|---------|---|
| ++ | incremento | x++ | a variável x é incrementada de 1 (é o mesmo que x=x+1) |
| -- | decremento | x-- | a variável x é decrementada de 1 (é o mesmo que x=x-1) |

Fonte: elaborado pela autora.

2.4.6 Conversões entre tipos de dados nas expressões

Nas expressões, muitas vezes, é necessário fazer conversões entre um tipo de dado e outro. Há, basicamente, dois tipos de conversões: implícita e explícita. Na conversão implícita, os dados são convertidos automaticamente, sem a intervenção do programador. Essa conversão automática pode ser realizada, entre os tipos abaixo, na direção das setas ou, ainda, na atribuição de constantes inteiras para tipos inteiros menores, desde que caiba no tipo menor:

double ← float ← long ← int ← short ← byte

Exemplos de conversão implícita:

- `long x = 12678` converte int para long
- `double dd = x` converte long para double
- `byte y = -2` converte int para byte
- `short s = y` converte byte para short
- `int i = y` converte short para int

Na conversão explícita (coerção ou cast) usa-se um operador unário de conversão, que consiste em colocar o identificador do tipo de destino

entre parênteses antes da expressão a ser convertida. Nesse caso, o programador deve estar ciente de que pode perder precisão, conforme o tipo de coerção.

Exemplos de conversão explícita:

- `double número = 15.99`
- `int n = (int)numero` converte double para int, perdendo a parte decimal.
- `double m = (double)n/10` converte n para double antes de efetuar a divisão para que seja realizada a divisão real e não a divisão inteira.
- `int num = (int)(numero / 2)` após efetuar a divisão real, converte o resultado da divisão para int, perdendo a parte decimal.

2.4.7 Precedência dos operadores

A maneira como uma expressão é avaliada (ou calculada) obedece às regras de precedência dos operadores. Podemos misturar operadores de vários tipos em uma mesma expressão e, nesse caso, devemos saber o que será calculado primeiro, ou seja, qual é a ordem de execução das operações dentro de uma mesma expressão. A linguagem Java obedece à tabela de precedência a seguir. Mas podemos utilizar parênteses para alterar a precedência de um operador, pois as operações incluídas nos parênteses mais internos são calculadas antes. Por exemplo, a multiplicação tem prioridade sobre a soma, mas, se colocarmos a soma entre parênteses, esta será calculada antes da multiplicação. Observe nos exemplos abaixo o uso dos parênteses e o uso de conversão explícita:

- `5 + 7 * 2` o resultado será 19
- `(5 + 7) * 2` o resultado será 24
- `6 + 4 / 2 + 3` o resultado será 11
- `(6 + 4) / (2 + 3)` o resultado será 2
- `3 / 2` o resultado será 1; foi realizada uma divisão de inteiros
- `(double) 3 / 2` o resultado será 1.5; foi realizada uma divisão de reais
- `3.0 / 2` o resultado será 1.5; foi realizada uma divisão de reais

- $9 / 2 + 1$ o resultado será 5
- $9 / (2 + 1)$ o resultado será 3
- $9 \% (2 + 1)$ o resultado será 0
- $5 + 3 * 2 != (5 + 3) * 2$ o resultado será true
- $5 + 3 * 2 >= (5 + 3) * 2$ o resultado será false
- $11 != (5 + 3) * 2 \&\& 5 + 3 * 2 == 16$ o resultado será false
- $11 != (5 + 3) * 2 || 5 + 3 * 2 == 16$ o resultado será true
- $(\text{int})(3.0 / 2) + 4$ o resultado será 5

Tabela 1 – Tabela de Precedência dos operadores

| Tabela de Precedência dos operadores | | |
|--------------------------------------|--------------------------------|--|
| | Operador | Descrição |
| 1º Máxima precedência | $\cdot \] [() (\text{tipo})$ | Separador, indexação, parâmetros, conversão de tipo. |
| 2º | $+ - \sim ! ++ -$ $-$ | <u>Operadores unários:</u> positivo, negativo, negação (inversão bit a bit), não lógico (Not), incremento, decremento. |
| 3º | $* / \%$ | Multiplicação, divisão e resto. |
| 4º | $+ -$ | Adição, subtração. |
| 5º | $<<$ $>>$ $>>>$ | Deslocamento (bit a bit) à esquerda, direita sinalizada, e direita não sinalizada (o bit de sinal será 0). |
| 6º | $<$ $<=$ $>=$ $<$ | <u>Operadores relacionais:</u> menor, menor ou igual, maior ou igual, maior. |
| 7º | $== !=$ | <u>Operadores de igualdade:</u> igual, diferente. |
| 8º | $\&$ | Operador lógico e bit a bit. |
| 9º | $^$ | Ou exclusivo (xor) bit a bit. |
| 10º | $ $ | Operador lógico ou bit a bit. |
| 11º | $\&\&$ | Operador lógico e (and) condicional. |
| 12º | $ $ | Operador lógico ou (or) condicional. |

| | | |
|-----|-------|-------------------------------------|
| 13º | ?: | Condisional: if-then-else compacto. |
| 14º | = op= | Operadores de atribuição |

Fonte: Deitel – Como programar.

CAPÍTULO 3

OS PRIMEIROS COMANDOS

Neste capítulo começaremos a apresentar os principais comandos das linguagens de programação de alto nível. Veremos os comandos que utilizam as unidades básicas do computador: a unidade de memória e as unidades de entrada e saída. Continuaremos utilizando a linguagem Java como instrumento de aprendizagem e, por isso, apresentaremos esses comandos em Java.

3.1 Comando de atribuição

Atua na memória do computador. É responsável por armazenar um valor em uma variável. Se a variável já possuía um valor, esse é perdido e o novo valor é gravado em cima.

3.1.1 Sintaxe

| |
|---|
| <aqui vai uma variável> = <aqui uma expressão>; |
|---|

3.1.2 Semântica

O computador executa esse comando da seguinte forma: avalia a expressão que aparece à direita do operador de atribuição e armazena o resultado na variável que aparece à esquerda do operador.

3.1.3 Exemplos

Tabela 2

| Comando | Explicação | Memória |
|---|--|---|
| int n1 = 13; | declara a variável n1 e armazena o valor 13 na variável declarada | <div style="display: flex; align-items: center; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">13</div> <div>n1</div> </div> |
| int resto = n1 % 5; | calcula o resto da divisão de 13 por 5 e armazena o resultado na variável resto, declarada no mesmo comando. | <div style="display: flex; align-items: center; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">3</div> <div>resto</div> </div> |
| Comando | Explicação | Memória |
| double salario = 1000.00; | declara a variável salário e armazena o valor 1000.00 na variável declarada. | <div style="display: flex; align-items: center; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">1000.00</div> <div>salario</div> </div> |
| double novoSal; novoSal = salario * 1.2; | o primeiro comando declara a variável novoSal e o segundo comando multiplica o valor armazenado na variável salario por 1.2 e obtém o resultado 1200.00. Este valor é então armazenado na variável novoSal | <div style="display: flex; align-items: center; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">1200.00</div> <div>novoSal</div> </div> |
| int val; val = n1; | o primeiro comando apenas declara a variável val. O segundo comando armazena o valor armazenado na variável n1, também na variável val | <div style="display: flex; align-items: center; justify-content: space-between;"> <div style="border: 1px solid black; padding: 2px;">13</div> <div>val</div> </div> |

Fonte: elaborada pela autora.

3.2 Saída de dados

É o envio de dados, mensagens etc. para uma unidade de saída do computador. Atela é usada como unidade padrão de saída. Os comandos de saída, também conhecidos como comandos de escrita, exibem na tela os itens especificados. Estes itens podem ser valores de variáveis, constantes, resultados de expressões e mensagens. Vários itens diferentes podem ser colocados em um mesmo comando de saída, onde devem ser concatenados com o operador +.

Um comando de saída diz ao computador para escrever ou exibir itens na tela. Em Java temos métodos que realizam essa tarefa. Para um programador iniciante, por enquanto, é indiferente se usaremos um método ou um comando, desde que obedeçamos à sintaxe e entendamos o que acontece quando o método ou o comando é executado.

3.2.1 Sintaxes

- a. `System.out.print (<aqui os itens a serem exibidos na tela>);`
- b. `System.out.println (<aqui os itens a serem exibidos na tela>);`
- c. `System.out.printf ("formatação"><aqui os itens a serem exibidos na tela>);`

Relação de alguns códigos de formatação (%) para serem usados no printf:

- %d – será substituído por um valor decimal (valores inteiros);
- %f – será substituído por um valor real (%8.2f reservando 8 posições da tela, das quais 2 serão usadas para as casas decimais);
- %c – será substituído por um caractere;
- %s – será substituído por uma cadeia de caracteres.

Alguns caracteres têm um significado especial, dentro de aspas, nos métodos de saída:

\n nova linha
\f nova tela
\t tabulação

3.2.2 Semântica

O computador executa esse comando da seguinte forma: exibe na tela cada um dos itens, buscando na memória os valores das variáveis, quando for o caso.

Na sintaxe a, exibe os itens e o cursor permanece na mesma linha.

Na sintaxe b, exibe os itens e o cursor vai para a próxima linha.

Na sintaxe c, é possível especificar o formato (*número de casas decimais e outros*) como os itens devem ser exibidos, aplicando os códigos de formatação.

Note que espaço em branco é um caractere, como qualquer outro, e deve ser especificado para que seja exibido na tela.

3.2.4 Exemplos

Supõe-se que as variáveis *som*, *media*, *a* e *b* já estão na memória, conforme figura abaixo. O comando de saída acessa a memória para obter os valores das variáveis.

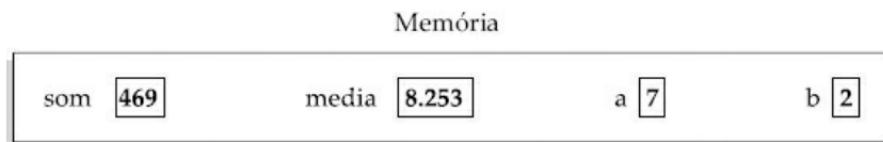


Figura 3

Fonte: elaborada pela autora.

- `System.out.print ("Saída de ");`
- `System.out.println("dados ");`
- `System.out.println ("Soma: " + som);`
- `System.out.println ("Resultado: "+a*b);`
- `System.out.printf ("Media: %.2f\n", media);`
- `System.out.printf ("Aidade do aluno %s é %d\n", "Carlos", 21);`

Saída de dados
Soma: 469

Resultado:14

Media: 8,25

Aidade do aluno Carlos é 21

3.3 Entrada de dados

É a captura de dados de uma unidade de entrada do computador. Usaremos o teclado como unidade de entrada padrão. Os comandos de entrada, também conhecidos como comandos de leitura, recebem os dados digitados no teclado, armazenando-os na memória.

Um comando de entrada diz ao computador para interromper a execução dos comandos e ler dados do teclado. Em Java, temos métodos que realizam essa tarefa, mas antes de examinarmos as classes da API vamos utilizar uma classe de nome Teclado, cujo código é dado abaixo. Essa classe contém muitos elementos avançados que fogem ao escopo de uma abordagem inicial, então ela deve ser copiada e compilada dentro do seu projeto, e, por enquanto, não devemos nos preocupar com detalhes de implementação. É suficiente conhecermos a interface pública da classe, ou seja, as assinaturas dos seus métodos públicos.

Repetindo o que já foi recomendado ao programador iniciante, no tópico de saída de dados, por enquanto, é indiferente se usaremos um método ou um comando, desde que obedeçamos à sintaxe e entendamos o que acontece quando o método ou o comando é executado.

3.3.1 Classe Teclado

```

import java.io.*;

/*
 * Classe que permite fazer leitura de dados do teclado,
 * com métodos estáticos.
 * Isto significa que não há necessidade de instanciar
 * um objeto para invocar os métodos.
 *
 * Sintaxe p/ chamada:
 * <nome_da_classe>.<nome_do_método>(<mensagem de solicitação>)
 *
 * Exemplo de chamada:
 * Teclado.leInt("Digite um número inteiro")
 */

public class Teclado {
    private static String s;
    private static InputStreamReader i = new InputStreamReader(System.in);
    private static BufferedReader d = new BufferedReader(i);

    /**
     * Lê um inteiro, exibindo na tela uma mensagem de solicitação.
     * @return int
     */
    public static int leInt(String msg) {
        int a = 0;
        System.out.print(msg);
        try {
            s = d.readLine();
            a = Integer.parseInt(s);
        } catch (IOException e) {
            System.out.println("Erro de I/O: " + e);
        } catch (NumberFormatException e) {
            System.out.println("deve ser digitado um inteiro: " + e);
        }
        return (a);
    }

    /**
     * Lê um double, exibindo na tela uma mensagem de solicitação.
     * @return double
     */
    public static double leDouble(String msg) {
        double a = 0;
        System.out.print(msg);
        try {
            s = d.readLine();
            a = Double.parseDouble(s);
        } catch (IOException e) {
            System.out.println("Erro de I/O: " + e);
        } catch (NumberFormatException e) {
            System.out.println("deve ser digitado um número: " + e);
        }
        return (a);
    }

    /**
     * Lê um string, exibindo na tela uma mensagem de solicitação.
     * @return String
     */
    public static String leString(String msg) {
        s = "";
        System.out.print(msg);
        try {
            s = d.readLine();
        } catch (IOException e) {
            System.out.println("Erro de I/O: " + e);
        }
        return (s);
    }

    /**
     * Lê um caractere exibindo na tela uma mensagam de solicitação.
     * @return Char
     */
    public static Character leChar(String msg) {
        s = "";
        System.out.print(msg);
        try {
            s = d.readLine();
        } catch (IOException e) {
            System.out.println("Erro de I/O: " + e);
        }
        return (s.charAt(0));
    }
}

```

1/fim da classe

3.3.2 Sintaxes

- para ler um número inteiro, `<var>` deve ser uma variável do tipo inteiro.

```
<var> = Teclado.leInt (<aqui a solicitação para que o usuário  
digite o valor>);
```

- para ler um número real, `<var>` deve ser uma variável do tipo real.

```
<var> = Teclado.leDouble (<aqui a solicitação para que o usuário  
digite o valor>);
```

- para ler um caractere, `<var>` deve ser uma variável do tipo caractere.

```
<var> = Teclado.leChar (<aqui a solicitação para que o usuário  
digite o caractere>);
```

- para ler uma sequência de caracteres, `<var>` deve ser uma variável do tipo String.

Obs.: ainda não vimos um tipo de dado capaz de armazenar uma sequência de caracteres. Estudaremos esse tipo mais adiante, por enquanto basta saber que o identificador desse tipo de dado é String.

```
<var> = Teclado.leString (<aqui a solicitação para que o usuário  
digite o string>);
```

3.3.3 Semântica

O computador executa esse comando da seguinte forma: antes de realizar a leitura, exibe na tela a solicitação para que o usuário digite o dado e espera até que o valor seja digitado no teclado; então, captura o valor e faz a atribuição para a variável que aparece à esquerda do operador de atribuição. Observe que a execução da entrada de dados está envolvendo as três unidades básicas: saída, entrada e memória, conforme

o exemplo abaixo.

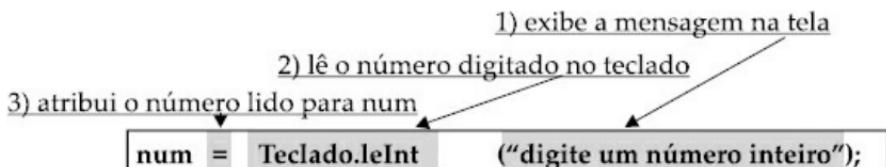


Figura 4

Fonte: elaborada pela autora.

Exemplos

- String n = Teclado.leString("Informe o nome: ");
- int id = Teclado.leInt("Digite a idade: ");

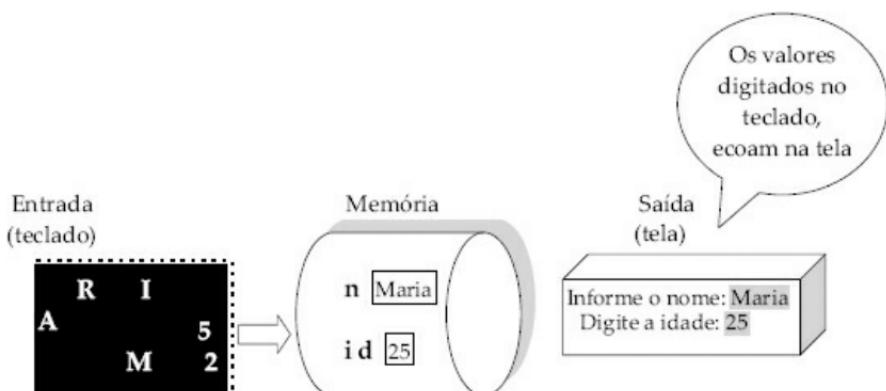


Figura 5

Fonte: elaborada pela autora.

3.4 Exemplo com entrada, atribuição e saída

O trecho de programa abaixo lê, do teclado, o salário de um funcionário, calcula um novo salário, para esse funcionário, com um reajuste de 5% e exibe na tela o novo salário.

```
double sal = Teclado.leDouble("Digite o salário:");
double novoSal = sal *1.05;
System.out.println("Novo salário:" + novoSal);
```

Este outro trecho de programa lê, do teclado, dois números inteiros, calcula a soma dos números lidos e exibe na tela a soma calculada.

```
int n1 = t.leInt("Digite um número inteiro: ");
int n2 = t.leInt("Digite outro número inteiro: ");
int soma = n1 + n2;

System.out.println("Soma dos números lidos: " + soma);
```

CAPÍTULO 4

CLASSES E OBJETOS

Este capítulo contém o foco principal de todo o nosso estudo, pois apresenta os conceitos fundamentais do paradigma orientação a objetos: classe e objeto.

Após o esclarecimento sobre os conceitos, daremos início à modelagem e à programação de uma classe, abordando todos os elementos básicos necessários ao programador iniciante, com explicações e exemplos.

4.1 Introdução

O conceito de classes e objetos é fundamental para o entendimento da tecnologia orientada a objetos. Nesse paradigma podemos dizer que projetar um software é modelar um conjunto de classes e seus relacionamentos, e uma aplicação em execução consiste em um conjunto de objetos que se comunicam.

A partir de agora, veremos que tudo são objetos: o aluno João da Silva é um objeto. A turma em que o aluno João matriculou-se é um objeto. O próprio ato de matrícula é um objeto.

4.2 Classes e objetos: conceitos fundamentais

Vamos começar examinando objetos do mundo real, tais como um telefone celular, um rádio, um carro etc.

4.2.1 Objetos do mundo real



Figura 6

Fonte: elaborada pela autora.

4.2.2 Estado e comportamento dos objetos

Vamos identificar dois aspectos dos objetos do mundo real:

Estado

Um conjunto de propriedades, relevantes ao contexto, que descrevem o objeto. É o conhecimento, ou seja, o que o objeto "conhece".

Comportamento

As ações realizadas pelo objeto ou as suas funcionalidades. O que o objeto "faz".

Os telefones têm estado (propriedades) – ligados ou desligados, modo (de chamada, ou outros) – e têm comportamento – recebe chamada, faz chamada, dispara alarme, envia mensagem, exibe menu.

Os rádios têm estado (propriedades) – ligado, desligado, volume atual, estação sintonizada – e têm comportamento – liga, desliga, aumenta o volume, diminui o volume, seleciona a estação.

Os carros têm estado (propriedades) – quilometragem, consumo, quantidade de combustível no tanque – e têm comportamento – anda, abastece, informa combustível, informa consumo.

4.2.3 Objetos

Observe que os objetos do mundo real variam quanto a sua complexidade, podendo necessitar de poucas ou muitas informações para representar o seu estado. Até mesmo outros objetos podem fazer parte do estado de algum objeto mais complexo.

Objetos de software são conceitualmente similares aos objetos do mundo real: têm estado e comportamento. Um objeto de software necessita armazenar apenas as propriedades que são relevantes ao contexto (ou negócio) que está sendo programado. Essas características são armazenadas em variáveis conhecidas como campos ou *atributos* (ou ainda, *variáveis de instância*) e determinam o estado interno do objeto. O comportamento (ou funcionalidades) dos objetos de software é expresso através de *métodos*.

4.2.4 Classes

Conforme podemos observar nos exemplos de objetos do mundo real, temos vários objetos do mesmo tipo. Observe que apresentamos no exemplo um conjunto de nove objetos, porém esses nove objetos pertencem a apenas três categorias ou tipos, pois temos três objetos do tipo telefone celular, quatro objetos do tipo rádio e dois objetos do tipo

carro. Essas categorias são o que chamamos de classes. Em termos de orientação a objeto dizemos que cada carro em particular é uma instância da classe **Carro**, assim como cada um dos três telefones pertencem à classe **Telefone Celular** e cada um dos quatro rádios é uma instância da classe **Radio**.

4.2.5 Diferença entre classe e objeto

Uma classe é um projeto ou um modelo ou uma especificação que define um tipo de objeto que pode ser construído. Um objeto, se e quando criado, é a concretização da especificação.

A classe pode ser comparada a uma receita de bolo. A partir da receita, podemos confeccionar vários bolos. Outra analogia que pode esclarecer a diferença é comparar uma classe a uma planta arquitetônica. A partir da mesma planta, poderemos construir várias casas, em locais diferentes, com diferentes cores, para diferentes proprietários.

A partir de uma classe, podem ser criados (instanciados) zero ou mais objetos.

vários objetos

uma classe: Calçado

atributos :

- modelo
- número
- cor



Figura 7 – Várias instâncias de uma classe.

Fonte: elaborada pela autora.

4.3 Linguagem de modelagem UML

É uma linguagem gráfica, independente da linguagem de programação, utilizada para modelar o software em desenvolvimento. Usaremos uma pequena parte da linguagem de modelagem *Unified*

Modeling Language (UML) para representar as classes e os objetos, através de diagramas e setas. Representaremos, também, alguns relacionamentos entre as classes, usando os recursos da linguagem UML.

A modelagem deve ser realizada antes da programação da classe e ajuda no entendimento do que deve ser codificado.

4.3.1 Diagramas de classes

As classes são representadas por um retângulo dividido em três segmentos: o segmento de nome, que conterá apenas o nome da classe que está sendo modelada; o segmento de atributos, que relacionará os atributos (isso é, as propriedades) que os objetos devem armazenar; e o segmento de serviços ou funcionalidades que a classe oferece, isso é, construtores e métodos.

Os atributos, em UML, são declarados com a sintaxe:

```
<visibilidade><nome> : < tipo >
```

Os métodos, em UML, são declarados com a sintaxe:

```
<visibilidade><nome do método> (<lista de parâmetros>):  
    <tipo de retorno>
```

A lista de parâmetros declara todos os parâmetros, separados por vírgulas com a sintaxe:

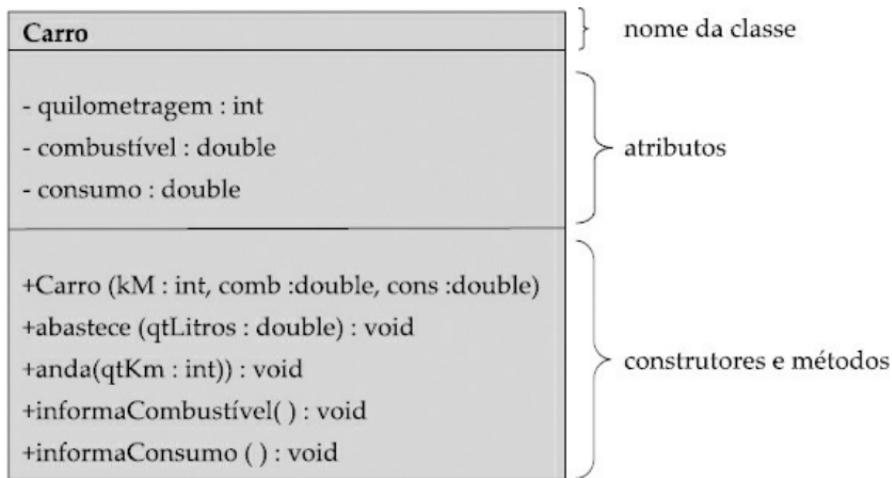
```
<nome> : < tipo >
```

Obs.: nos tópicos seguintes veremos cada um desses elementos (parâmetro, retorno etc.) detalhadamente. Por enquanto, preocupe-se apenas com as partes que compõem o diagrama.

Indicamos a visibilidade dos membros de uma classe usando os sinais:

- + indica visibilidade externa total (public)
- indica nenhuma visibilidade externa (private)
- # indica visibilidade externa limitada (protected)

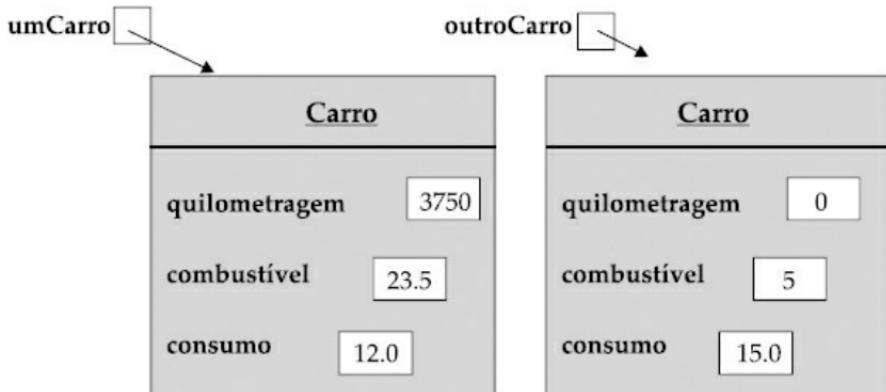
Exemplo: diagrama da classe Carro



4.3.2 Diagramas de objetos

Os objetos em UML são representados por um retângulo dividido em dois segmentos: o segmento de nome, que conterá apenas o nome da classe (sublinhado) à qual o objeto pertence, e o segmento de atributos, que mostrará possíveis valores dos atributos do objeto em um dado momento. Os diagramas dos objetos podem ser utilizados para que visualizemos a memória do computador, onde os objetos serão armazenados quando criados.

Exemplo: diagramas de objetos da classe Carro



4.3.3 Relacionamentos entre as classes

Usam-se diferentes tipos de setas para representar os diferentes tipos de relacionamentos entre classes.

-----> Dependência

-----> Associação

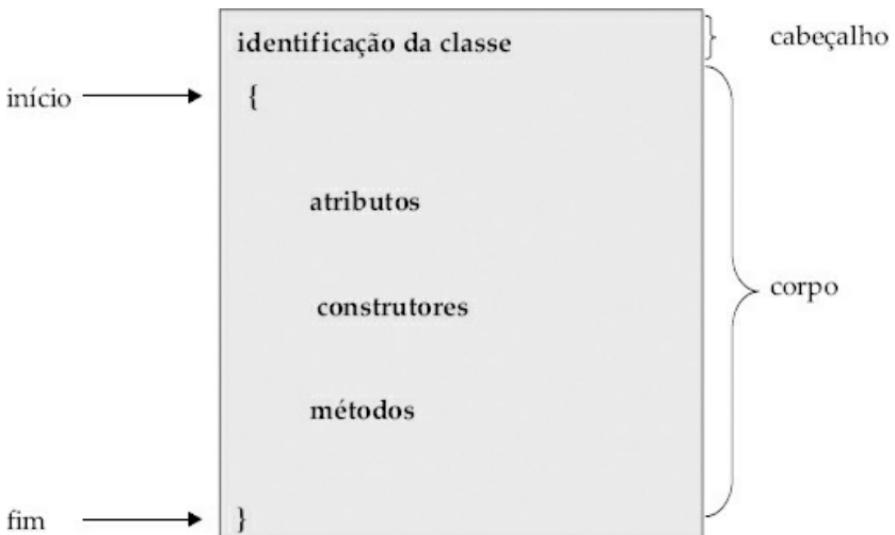
-----> Herança

4.4 O que uma classe deve conter em Java

A classe contém o modelo de um objeto, definindo seus métodos (comportamento) e os dados que o objeto pode armazenar (estado). A classe determina como o objeto deve ser criado através de um construtor de objetos.

Em Java, após especificar o cabeçalho da classe, contendo a sua

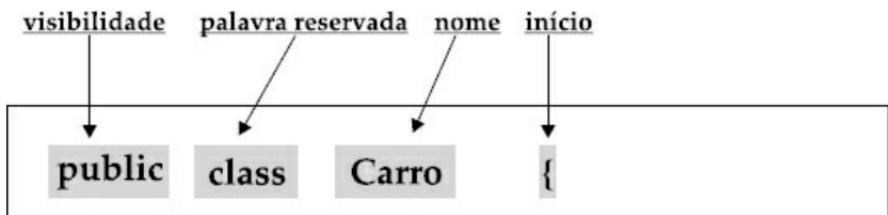
identificação e o seu início com um “abre-chave”, temos o corpo da classe, constituído, basicamente, de três partes: os atributos, os construtores e os métodos. Encerramos a classe com um “fecha-chave”.



4.4.1 Identificação da classe

O cabeçalho da classe contém o nome (identificador) da classe e deve ser precedido pela palavra reserva da classe e pela visibilidade da classe (public, private etc.). Após a identificação, coloca-se um “abre-chave” (na mesma linha ou na próxima), indicando o início da classe.

Exemplo da identificação da classe Carro:



4.4.2 Atributos

Os atributos, também denominados campos ou variáveis de instância, armazenam as propriedades dos objetos. Normalmente, são definidos como privados, ou seja, seus valores não podem ser modificados fora da classe.

Os identificadores (nomes) dos atributos são criados pelo programador. O tipo do atributo (int, double, etc.) indica o conteúdo que pode ser armazenado e o espaço de memória necessário.

As propriedades dos objetos podem ser valores simples representados pelos tipos primitivos da linguagem ou podem ser mais complexas, e então, representadas por tipos-objeto. Nesse caso diz-se que há um relacionamento de associação entre a classe que tem um atributo cujo tipo é um objeto e a classe que é o tipo deste atributo.

Atente para a diferença entre um atributo e os valores que ele pode assumir. Todos os objetos da mesma classe têm os mesmos atributos, porém diferentes valores armazenados. O próprio objeto tem os valores dos seus atributos alterados durante a sua existência, caracterizando uma mudança de estado, já que o estado de um objeto é determinado pelas suas propriedades. Por exemplo, um objeto carro que está com quinze litros de combustível no tanque, ao ser abastecido com mais dez litros, mudou de estado, pois o valor armazenado no atributo *combustível* mudou de 15 para 25.

Exemplo da declaração de atributos da classe Carro:

| visibilidade | tipo do atributo | nome do atributo | comentário |
|--------------|------------------|------------------|--|
| private | int | quilometragem; | //armazena a quilometragem atual do carro |
| private | double | combustível; | //armazena a quant. de combustível no tanque |
| private | double | consumo; | //armazena quantos quilômetros o carro faz por litro |

4.4.3 Métodos

Os métodos atuam sobre os objetos e são responsáveis pela realização das tarefas. São os serviços oferecidos pelos objetos de uma determina da classe.

A visibilidade determina se o método poderá ser acessado externamente, ou seja, se outras classes poderão chamar (invocar) o método. A visibilidade pode ser pública, privada ou protegida. Usaremos, em geral, *public* ou *private*. A maioria dos métodos das classes que apresentaremos são públicos.

Dentro dos métodos, mais especificamente no corpo do método, é onde colocamos os comandos que serão executados quando o método for chamado. Observe que os comandos realizam a tarefa pela qual o método é responsável. Um método somente será executado quando invocado (chamado). A chamada de um método ocorre através da execução de um comando de chamada que está dentro de outro método. Enquanto um método está sendo executado é ele que detém o chamado “controle de execução”. Então, quando um método é chamado, ele recebe o controle, e quando termina sua execução, ele devolve o controle de volta para o método que o chamou.

Os métodos são definidos para dotar as classes de certas funcionalidades. Por exemplo, os objetos da classe Carro devem oferecer a possibilidade de colocar combustível no tanque, sendo dotados de um método para realizar essa tarefa. Outro exemplo: vamos imaginar, sem nos preocuparmos com detalhes, uma classe ContaBancaria com apenas dois atributos, o número da conta e o saldo. É natural que a classe ofereça a possibilidade de fazer depósitos e saques e ainda consulta ao saldo. Esses serviços devem ser realizados por métodos desenvolvidos com essas finalidades. Vamos examinar, com mais cuidado, dois desses métodos. Primeiramente, o método *fazDepósito*, que tem como responsabilidade alterar o saldo da conta bancária, acumulando no saldo o valor que está sendo depositado. Mas como é que o método “sabe” qual é o valor a ser depositado? Esse valor será sempre o mesmo? É claro que não. O método não tem como prever ou calcular esse valor. Então, o método deve receber esse valor quando for chamado para ser executado, e cada vez que for chamado pode receber um valor diferente. Logo, o método deve declarar essa necessidade de receber um valor externo e o tipo do valor, isso é, o método deve declarar um parâmetro. Assim como esse método precisou de apenas um parâmetro, outros métodos podem necessitar de vários valores externos, e por isso a declaração da lista de parâmetros, que pode estar vazia, caso o método não precise receber nenhum valor. Esse mecanismo é conhecido como passagem de parâmetros.

Observe que o método *fazDeposito*, quando executado, altera o estado do objeto, pois modifica o valor armazenado em um de seus atributos.

Agora vamos analisar o método *informaSaldo*, que tem como responsabilidade exibir o saldo da conta bancária na tela. O método “sabe” qual é o valor do saldo? O método tem acesso ao saldo, pois é um atributo da classe, então todos os objetos, ou seja, todas as contas bancárias armazenam esse valor. Logo, esse método não precisa de parâmetros e deve declarar uma lista de parâmetros vazia.

Observe que o método *informaSaldo* não altera o estado do objeto, pois apenas acessa o valor armazenado em um de seus atributos.

Os dois métodos analisados acima foram projetados e desenvolvidos de forma a não retornar ou devolver qualquer valor, e por isso o tipo do retorno deve ser a palavra reservada *void*, que indica a ausência de retorno. Mas em muitos casos é necessário que o método retorne algum valor, ou até mesmo um objeto, ou seja, a chamada do método produz um “resultado” que será recebido pelo chamador. Então, o tipo do valor retornado deve aparecer na declaração do método em vez da palavra *void*. Por exemplo, a classe Carro poderia ter um método que calculasse quantos quilômetros o carro consegue andar somente com o combustível que tem no tanque, sem abastecer. Então, esse método deveria retornar esse valor, e isso seria indicado colocando *int* em vez de *void*. No corpo do método, além dos comandos que fazem o cálculo, deve ter um comando para enviar esse valor para fora do método. Esse comando é conhecido como comando de retorno e tem a seguinte sintaxe:

palavra-chave especifica o valor que o método devolve

↓

return

↓

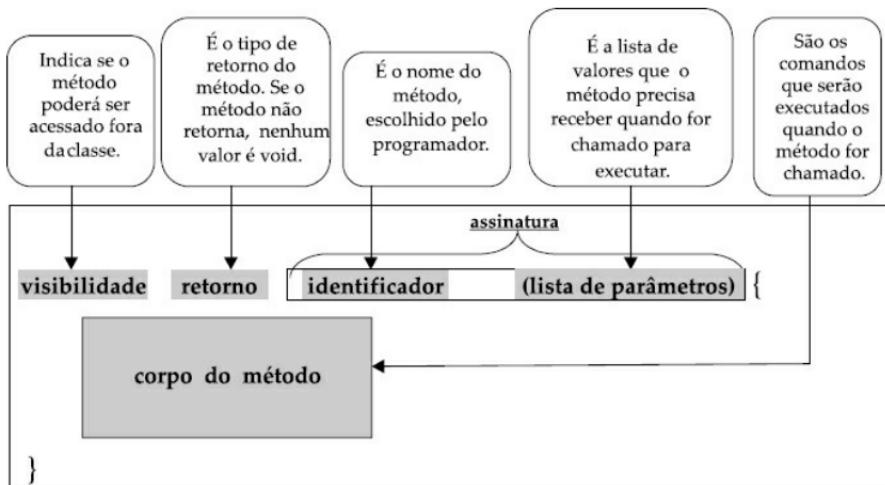
<expressão> ;

Quando esse comando é atingido, o controle de execução sai imediatamente do método e volta para o método que o chamou, levando o valor.

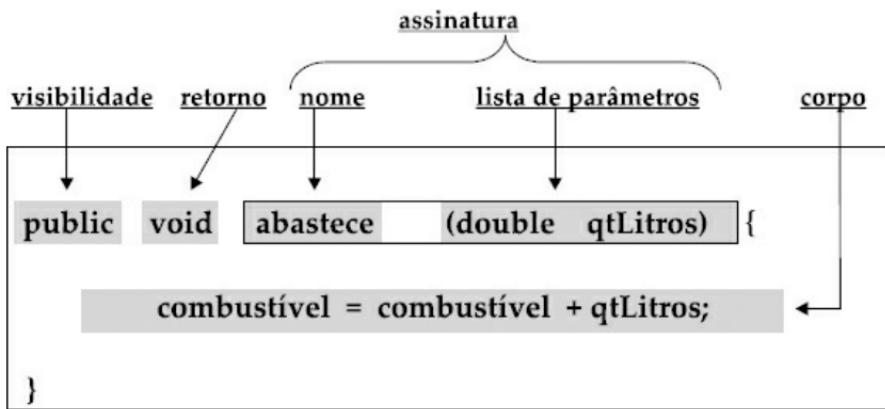
Os métodos têm o nome criado pelo programador. O nome deve ser cuidadosamente escolhido pelo programador, de forma que o nome do método indique o que o método faz. Podemos ter, propositalmente, vários métodos com o mesmo nome em uma mesma classe. Nesse caso,

dizemos que esses métodos foram sobrecarregados. A sobrecarga consiste em termos métodos com nomes iguais, mas lista de parâmetros diferentes para que o compilador decida qual dos métodos será acionado.

Na figura abaixo, podemos observar que os métodos têm duas partes: o cabeçalho (constituído pela visibilidade, pelo tipo de retorno e pela assinatura (nome + parâmetros)) e o corpo (constituído pelas instruções ou comandos da linguagem).



Exemplo de um método da classe Carro:



Visibilidade – A palavra reservada *public* significa que o método poderá ser acessado externamente, ou seja, outras classes poderão chamar (invocar) o método.

Nome – É o identificador do método.

Retorno – É o tipo de retorno do método. Esse método não retorna nenhum valor, e por isso o uso da palavra reservada *void*.

Lista de parâmetros – Esse método necessita de uma informação externa (parâmetro) para realizar esta tarefa. Quando esse método for invocado, deve receber um valor do tipo *double*, que é a quantidade de combustível que será acrescentada no tanque.

Corpo – O corpo do método contém os comandos que serão executados quando o método for chamado. Nesse caso, é necessário apenas um comando para abastecer o carro.

4.4.4 Construtores

Os construtores inicializam os atributos do objeto quando o objeto é instanciado pelo operador *new*.

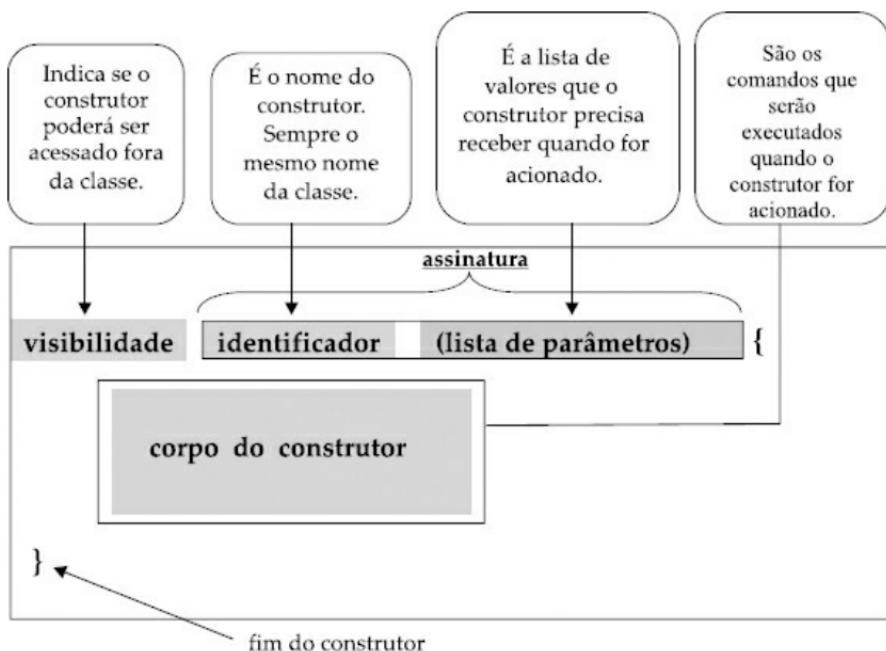
Um construtor é um tipo especial de método que configura os atributos de um objeto, ou seja, coloca valores nos campos quando um objeto é criado. Esses valores podem ser calculados, fixos ou recebidos via parâmetros, isso é, quando o construtor é ativado (pelo operador *new*) são enviados os valores necessários para armazenar nos atributos do objeto que está sendo criado. Exemplo: quando houver necessidade de

criar (instanciar) um objeto da classe Carro, normalmente fora da própria classe, será acionado o construtor e este deverá receber os seguintes valores que devem ser armazenados no objeto que está sendo criado: a quilometragem, o consumo e a quantidade de combustível no tanque.

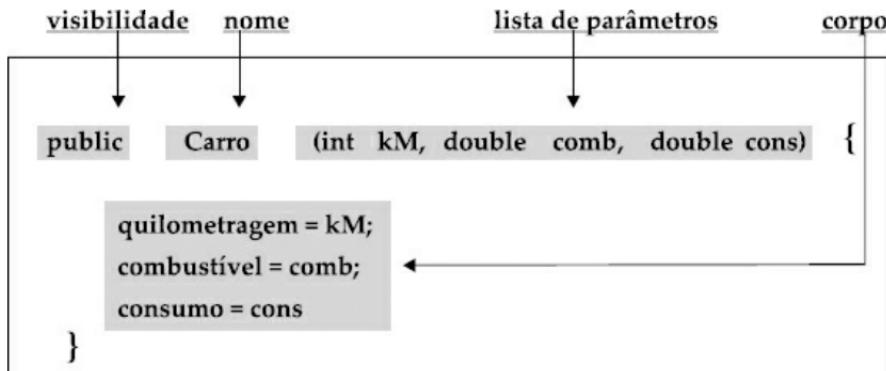
A visibilidade determina se o construtor poderá ser acessado externamente, ou seja, se outras classes poderão instanciar objetos dessa classe usando este construtor. Os construtores das classes que apresentaremos são todos públicos. Têm sempre o mesmo nome que a classe, podendo haver vários em uma mesma classe (sobrecarga). Nesse caso, são identificados pela quantidade, posição e tipo de parâmetros.

Toda a classe tem, pelo menos, um construtor. Se o programador não colocar nenhum construtor na classe, o compilador fornece o chamado construtor padrão, que é um construtor sem parâmetros e cuja tarefa é configurar os atributos do objeto que está sendo construído com valores padrão (zero para tipos numéricos primitivos e null para tipos-referência). Isso impede que algum objeto seja criado em um estado inconsistente.

Na figura abaixo, podemos observar que os construtores têm duas partes: o cabeçalho (constituído pela visibilidade e pela assinatura (nome + parâmetros)) e o corpo (constituído pelas instruções, ou seja, comandos da linguagem Java).



Exemplo de um construtor da classe Carro:



Visibilidade – A palavra chave `public` significa que o construtor

poderá ser acessado externamente, ou seja, outras classes poderão instanciar objetos da classe Carro.

Nome – Carro é o identificador do construtor, pois deve ter o mesmo nome da classe.

Lista de parâmetros – Refere-se a uma lista de variáveis com seus tipos, indicando que o construtor precisa receber valores para armazenar nessas variáveis. Quando o construtor for acionado, deverão ser passados valores (argumentos) adequados a cada uma dessas variáveis. O construtor deve receber um int, um double e um double, nessa ordem.

Corpo – O corpo do construtor contém os comandos que serão executados quando o construtor for acionado. Observe que os comandos configuram os atributos do objeto, armazenando neles os valores recebidos via parâmetros.

4.5 Desenvolvimento da classe Pessoa, passo a passo

Vamos supor que um cliente encomende um software para uma empresa. A aplicação envolve várias classes, entre elas a classe Pessoa, que deve ser responsável pelo conhecimento e comportamento, descritos abaixo.

Por enquanto, os requisitos para projetar a classe Pessoa são: que os objetos dessa classe armazenem o nome e a idade da pessoa, que seja oferecida uma funcionalidade para alterar a idade da pessoa quando ela faz aniversário e também que a classe ofereça a possibilidade de exibir na tela o nome e a idade da pessoa. Posteriormente, acrescentaremos outras funcionalidades.

Essa classe, considerando apenas os requisitos acima mencionados pode ser desenvolvida por um programador iniciante, com o conhecimento adquirido até o momento.

– Passo 1:

Antes de começarmos a desenvolver uma classe, devemos ler atentamente a especificação para identificarmos os atributos (com seus tipos), a forma como os objetos devem ser configurados pelo construtor e os métodos que devem estar presentes na classe. Para identificarmos os atributos precisamos responder à seguinte pergunta:

- Quais são os dados que os objetos da classe que estamos desenvolvendo devem armazenar?

Na classe que estamos analisando os dados são o nome e a idade. Sabemos que a idade é um número inteiro, então devemos declarar um atributo do tipo int para armazenar a idade. Mas ainda não vimos um tipo de dado capaz de armazenar um nome, que é uma sequência de caracteres. Estudaremos esse tipo mais adiante, por enquanto basta saber que o identificador desse tipo de dado é String, e então declaramos um atributo do tipo String para armazenar o nome da pessoa. Os atributos, em geral, devem ser privados.

Aqui podemos fazer a primeira parte do diagrama de classe e alguns diagramas de objetos, com dados fictícios para visualizarmos os objetos na memória quando forem construídos.

– Passo 2:

Para o construtor precisamos visualizar o objeto na memória e responder à seguinte pergunta:

- Quais são os dados que o construtor deve receber para inicializar os campos do objeto?

O que não se sabe e não pode ser calculado deve ser recebido via parâmetro. Então, nesse caso, o construtor deve receber o nome e a idade da pessoa para preencher os respectivos atributos.

– Passo 3:

Para os métodos precisamos visualizar o objeto na memória, sendo afetado ou não pelo método em questão, e responder a duas perguntas:

- O método deve retornar algum resultado?

Se sim, o tipo do valor a ser retornado deve aparecer no cabeçalho do método, do contrário coloca-se a palavra reservada void.

- Para realizar a sua tarefa o método precisa de algum valor externo?

Se sim, deve receber esse(s) valor(es) via parâmetros. Não podemos esquecer que o método tem acesso aos atributos do objeto da classe à qual pertence, ou seja, valores internos.

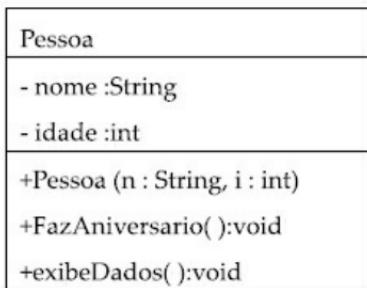
O corpo do método vai depender da tarefa a ser realizada. Aqui é necessária a lógica de programação para especificar os comandos que devem ser executados quando o método for chamado. Se o método retorna algum resultado, deve haver um comando que realize o retorno.

– Passo 4:

Devemos, primeiramente, fazer a modelagem da classe (os diagramas UML), para depois escrevermos a classe em Java. Observe que fica mais fácil se, à medida que formos respondendo as perguntas acima descritas e relendo a especificação, formos preenchendo os diagramas, com os elementos já identificados.

em UML

Diagrama de classe:



Diagramas de objetos:



```

public class Pessoa {//início da classe
    private String nome;
    private int idade;
    public Pessoa (String n, int i) {
        nome = n;
        idade = i;
    }
    public void faz aniversário() {
        idade = idade + 1;
    }
    public void exibeDados( ) {
        System.out.println (nome);
        System.out.println (idade);
    }
} // fim da classe
  
```

4.6 Instanciação (criação) de objeto

Um objeto só existe se for instanciado com o operador new.

A criação (instanciação) de um novo objeto em Java é realizada pelo operador *new*. Após a instanciação, o objeto passa a ocupar um espaço na memória, e o endereço desse objeto deve ser armazenado em uma variável para que o objeto possa ser acessado. A variável que é capaz de armazenar uma referência de um objeto é conhecida como variável-referência ou variável-objeto. Conforme já mencionado anteriormente,

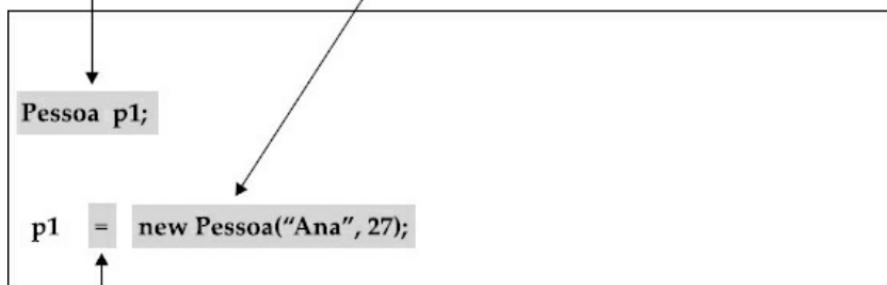
essa variável não armazena o objeto, e sim o seu endereço ou referência. Costumamos dizer que a variável aponta para o objeto e, muitas vezes, esse tipo de variável é denominada *pointer*, ou ponteiro. Graficamente, usamos setas para representar o conteúdo das variáveis-referência. Quando essa variável não contém o endereço de nenhum objeto, o seu valor é *null*.

Embora a classe Pessoa tenha sido desenvolvida (codificada, compilada e testada), isso por si só não faz com que exista algum objeto dessa classe. Os objetos não existem se não forem explicitamente criados (instanciados). Vimos que a classe Pessoa tem um construtor que serve para configurar os atributos de um objeto. Esse construtor só será executado se for acionado pelo operador new, que é responsável por alocar o objeto na memória e por acionar o construtor.

Exemplo da criação de um objeto em três passos

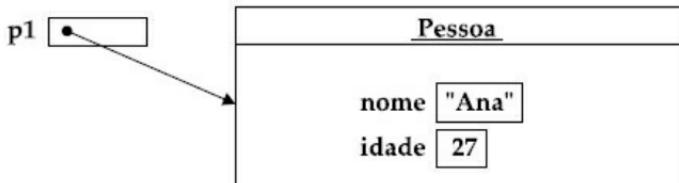
1) declara a variável objeto de nome p1 do tipo pessoa.

2) aloca área de memória para o novo objeto e aciona o construtor, enviando os valores.



3) atribui a p1 o endereço alocado.

Na memória:

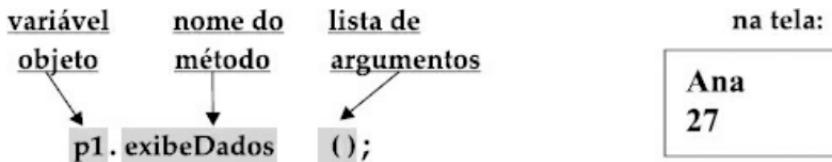


4.6.1 Chamada (invocação) de método

Os métodos somente serão executados quando chamados, e essa chamada pode ser feita de dentro de outro método da mesma classe ou de fora da classe. Os métodos, quando não são estáticos, atuam sobre os objetos da classe à qual pertencem e, nesse caso, para fazer uma chamada de método fora da sua classe é necessário termos um objeto instanciado. Então, chamar um método consiste em enviar uma referência do objeto sobre o qual o método deve atuar e uma lista de argumentos que combine com a lista de parâmetros da declaração do método chamado. A

classe Pessoa definiu alguns métodos. Uma vez criado o novo objeto (referenciado por p1), podemos aplicar métodos da classe Pessoa a esse objeto.

4.6.2 Exemplos de chamadas de métodos



4.6.3 O que acontece na memória quando um método é chamado (invocado) de fora da classe onde se encontra

Considere os comandos abaixo, fora da classe Pessoa:

```
//declaração de variáveis-objeto
Pessoa pA, pB, pC;

//instanciação de objetos
pA = new Pessoa ("Laura", 21);

pB = new Pessoa ("Carlos", 33);

pC = new Pessoa ("Paula", 60);
```

Memória, após a execução dos comandos acima:

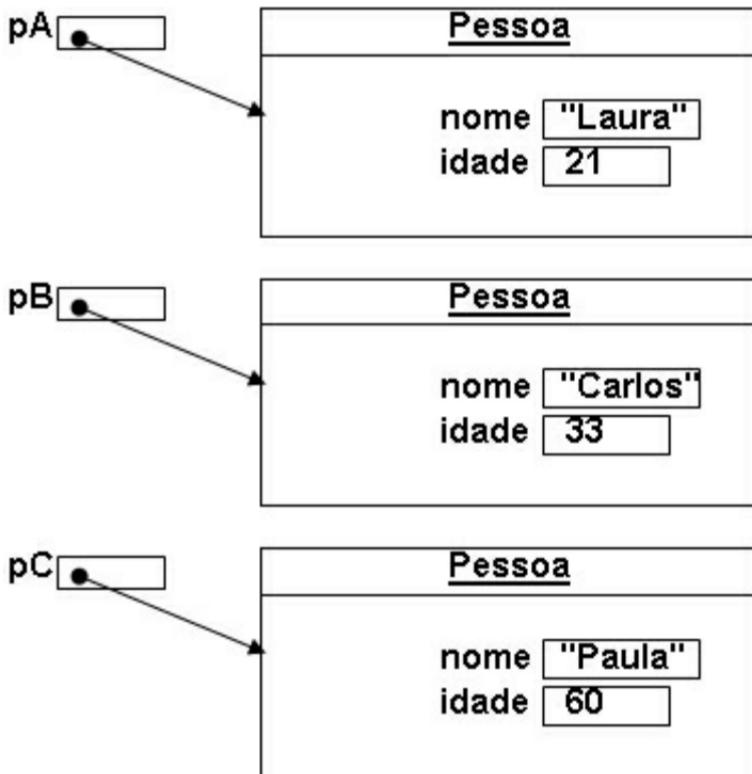


Figura 8 – Os objetos instanciados.
Fonte: elaborada pela autora.

```
//chamada do método para o objeto apontado por pB
pB.fazAniversario();
```

Memória, durante a execução do método fazAniversario, após a execução do comando de atribuição:

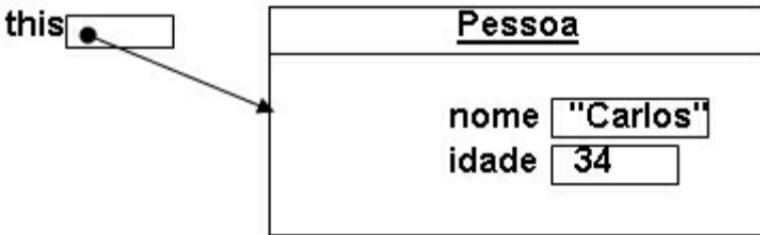


Figura 9 – O objeto alterado pelo método executado.

Fonte: elaborada pela autora.

Observe que o método foi chamado para o objeto cuja referência está armazanada na variável *pB*. Essa variável existe somente no trecho de código onde foi declarada. Então, essa referência é recebida em *this* (este objeto) no método *fazAniversario()*.

O conteúdo de *pB* é copiado para *this*. Então *this* referencia o objeto corrente (aquele que chamou o método que está sendo executado). Se o programador quiser deixar claro, dentro dos métodos, que está se referindo aos atributos *nome* e *idade* e não a qualquer outra variável, pode, explicitamente, usar a referência *this*. No caso da classe *Pessoa* é opcional, pois não há outras variáveis nos métodos nem no construtor com o mesmo nome dos atributos. Se houvesse, então seria necessário o uso explícito da referência *this* para distinguir os atributos das outras variáveis.

Alguns programadores adotam a prática de sempre se referir aos atributos com a referência *this* explícita, seja ou não necessário.

A figura abaixo mostra como ficaria a classe *Pessoa*, usando explicitamente a referência *this*:

```
public class Pessoa {  
    private String nome;  
    private int idade;  
  
    public Pessoa (String n, int i) {  
        this.nome = n;  
        this.idade = i;  
    }  
  
    public void fazAniversario( ) {  
        this.idade = this.idade + 1;  
    }  
  
    public void exibeDados( ) {  
        System.out.println (this.nome);  
        System.out.println (this.idade);  
    }  
}
```

A figura abaixo mostra como ficaria a classe Pessoa, usando, obrigatoriamente, a referência this no construtor, pois há duas variáveis com os mesmos nomes dos atributos. Nos outros métodos o uso do this é opcional.

```
public class Pessoa {  
    private String nome;  
    private int idade;  
  
    public Pessoa (String nome, int idade) {  
        this.nome = nome  
        this.idade = idade;  
    }  
  
    public void fazAniversario( ){  
        this.idade = this.idade + 1;  
    }  
  
    public void exibeDados( ){  
        System.out.println (this.nome);  
        System.out.println (this.idade);  
    }  
}
```

CAPÍTULO 5

MÉTODOS DE CONFIGURAÇÃO E ACESSO AOS ATRIBUTOS

Neste capítulo veremos dois tipos de métodos que são utilizados em grande parte das classes. A comunidade de desenvolvedores Java estabeleceu algumas convenções para esses métodos, que devem ser adotadas pelos programadores. São os famosos métodos `get` e `set`.

5.1 Introdução

Embora o programador tenha liberdade para criar os identificadores (nomes) dos métodos, existem algumas convenções que são seguidas pela comunidade de desenvolvedores Java quando se tratarem de métodos de acesso aos atributos e de métodos de configuração de atributos.

Normalmente, os atributos de uma classe são declarados privados. Isso significa que fora da classe não temos acesso, isso é, não podemos nos referirmos diretamente a esses atributos. Então, esse acesso deve ser feito via método, seja para obter o valor do atributo, seja para armazenar um valor em um atributo.

Quando for conveniente, a classe deve oferecer essas funcionalidades, através dos métodos conhecidos como `getter's` e `setter's`.

5.2 Métodos de acesso aos atributos

Um método de acesso a um atributo é um método que, quando acionado, retorna o valor armazenado no atributo. Costuma-se usar a palavra `get` seguida do nome do atributo para compor o identificador do método de acesso. No corpo do método devemos ter o comando `return`, responsável pelo retorno do valor.

Obs.: se o atributo for do tipo boolean, o método de acesso pode iniciar por `get` ou `is`.

5.2.1 Exemplos de métodos get

- Método da classe Pessoa que retorna o nome da pessoa:

```
public String getNome( ) {  
    return nome;  
}
```

- Método da classe Pessoa que retorna a idade da pessoa:

```
public int getIdade( ) {  
    return idade;  
}
```

- Método da classe Livro que retorna o valor do atributo emprestado (true ou false):

```
public boolean isEmprestado( ) {  
    return emprestado;  
}
```

5.3 Métodos de configuração de atributos

Um método de configuração de um atributo é um método que é acionado para enviar um valor para ser armazenado no atributo. Costuma-se usar a palavra `set` seguida do nome do atributo para compor o identificador do método de configuração. No corpo do método devemos ter um comando de atribuição, responsável pelo armazenamento do valor no

atributo. Quando necessário, podemos ainda ter um comando (if) que verifica a validade do valor a ser armazenado no atributo, impedindo, dessa forma, que o atributo armazene um valor inválido, deixando o objeto em um estado inconsistente.

Obs.: o comando if será abordado no Capítulo 8.

5.3.1 Exemplos de métodos set

- Método da classe Pessoa que configura o nome da pessoa:

```
public void setNome(String n ) {  
    nome = n;  
}
```

- Método da classe Pessoa que configura a idade da pessoa:

```
public void setIdade(int id) {  
    idade = id;  
}
```

- Método da classe Pessoa que configura a idade da pessoa mas não permite idade negativa nem zero:

```
public void setIdade(int id) {  
    if (id > 0)  
        idade = id;  
}
```

- Método da classe ContaBancaria que configura o saldo inicial da conta, mas não permite valor negativo:

```
public void setSaldo(double valor) {  
    if (valor >= 0)  
        saldo = valor;  
}
```

- Método da classe Data que configura o mês, mas não permite mês fora do intervalo 1 a 12. Em caso de mês inválido, configura com 1:

```
public void setMes (int umMes) {  
    if (umMes >= 1 && umMes <= 12)  
        mes = umMes;  
    else mes = 1;  
}
```

5.4 A classe Pessoa com os métodos get e set, sem usar a referência this

```
public class Pessoa {  
    //atributos  
    private String nome;  
    private int idade;  
  
    //construtor  
    public Pessoa (String n, int i){  
        setNome( n );  
        setIdade( i );  
    }  
  
    //métodos  
    public void setNome(String n) {  
        nome = n;  
    }  
  
    public String getName() {  
        return nome;  
    }  
    public void setIdade(int id) {  
        idade = id;  
    }  
    public int getIdade() {  
        return idade;  
    }  
    public void fazAniversario() {  
        idade = idade + 1;  
    }  
  
    public void exibeDados(){  
        System.out.println ("Nome: " + getName());  
        System.out.println ("Idade: " + getIdade());  
    }  
} //fim da classe
```

5.5 A classe Pessoa com os métodos get e set, usando a referência this

```
public class Pessoa {  
    //atributos  
    private String nome;  
    private int idade;  
  
    //construtor  
    public Pessoa (String n, int i) {  
        setNome( n );  
        setIdade( i );  
    }  
  
    //métodos  
    public void setNome(String n) {  
        this.nome = n;  
    }  
  
    public String getNome() {  
        return this.nome;  
    }  
  
    public void setIdade(int id) {  
        this.idade = id;  
    }  
  
    public int getIdade() {  
        return this.idade;  
    }  
  
    public void fazAniversario() {  
        this.idade = this.idade + 1;  
    }  
  
    public void exibeDados() {  
        System.out.println ("Nome: " + getNome());  
        System.out.println ("Idade: " + getIdade());  
    }  
}//fim da classe
```


CAPÍTULO 6

CLASSE DE TESTE

Neste capítulo veremos como testar as classes utilizando outra classe, a chamada classe de teste, contendo o método main.

6.1 Introdução

Uma aplicação consiste em um código executável pelo computador. Uma aplicação Java consiste em um código executável pela Java *Virtual Machine* (JVM). Esse código é gerado pelo compilador a partir do chamado código-fonte que foi codificado pelo programador. Todo o código Java é adequadamente dividido em um conjunto de classes. Cada classe tem suas responsabilidades. Por exemplo, a responsabilidade da classe Pessoa (Capítulo 5) é oferecer a possibilidade de serem instanciados objetos Pessoa e oferecer os serviços que podem ser realizados por esses objetos. Mas nada acontece se não houver o acionamento do construtor para que um objeto seja construído e se não houver a chamada dos métodos para atuarem sobre o objeto criado. E tudo isso costuma ser realizado em outras classes que se relacionam com a classe Pessoa e funcionam como clientes da classe, utilizando seus serviços. A classe Pessoa, por si só, não faz nada, pode apenas ser compilada, mas não pode ser executada. Para que uma classe possa ser executada pela JVM, precisa ter um método especial, que é o método principal a partir do qual inicia a execução. Esse método é conhecido como método main e tem uma assinatura especial.

A aplicação deve ser construída de forma que uma única classe, do conjunto de classes que se relacionam, contenha o método main, e então essa classe pode ser executada pela JVM e pode disparar a execução do software.

6.2 Em que consiste uma classe de teste

Frequentemente, teremos a necessidade de testar uma única classe, e para isso construiremos uma outra classe, cuja única finalidade é servir como classe de teste, ou seja, será dotada de um método main

para disparar a execução. O corpo do método main conterá os comandos necessários para instanciar objetos e chamar os métodos da classe que estamos testando.

Uma classe de teste não tem atributos e pode ser constituída de um único método de nome *main*, que tem a seguinte assinatura:

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

A classe de teste pode conter também alguns métodos auxiliares, que são chamados de dentro do método main.

O corpo do método main deve conter comandos para instanciar objetos da classe que estamos testando e comandos para chamar os métodos da classe sendo testada, para verificar se eles funcionam.

6.3 A classe TestePessoa

```
public class TestePessoa
{
    public static void main (String args[])
    {
        Pessoa p = new Pessoa("Maria" , 25);
        p.exibeDados( );
        p.fazAniversario( );
        p.exibeDados( );
    } // fim do método main
} // fim da classe de teste
```

Quando mandamos executar a classe *TestePessoa*, a JVM começa executando o primeiro comando do método main dessa classe, o qual cria o objeto na memória e aciona o construtor da classe *Pessoa*, enviando os dados "Maria" e 25. O endereço do objeto é atribuído para a variável *p*. O próximo comando a ser executado é a chamada do método *exibeDados* para o objeto apontado por *p*. Esse método exibe os dados da pessoa na tela. Logo após, é feita a chamada e executado o método *fazAniversario*, o qual altera a idade da pessoa na memória. Para testar se a idade foi

devidamente alterada há uma nova chamada ao método `exibeDados`, que agora deve exibir 26 para a idade da pessoa, comprovando que o método `fazAniversario` está funcionando.

Note que os métodos da classe `Pessoa` só foram executados porque foram chamados, e é assim que funciona: os métodos só são executados quando são chamados, e o construtor só é executado quando for acionado por um operador `new`.

6.4 Outra classe TestePessoa

Aqui ampliamos a classe de teste, colocando mais comandos dentro do método `main`, instanciando mais objetos e acionando os métodos sobre esses novos objetos, que são identificados pelas variáveis de referência `p1`, `p2` e `p3` declaradas do tipo `Pessoa`, apontando para os objetos instanciados. Todos acionam o mesmo construtor, porém com parâmetros diferentes.

```
1 public class TestePessoa{  
2     public static void main( String[] args) {  
3  
4         //Declaração de variáveis objeto  
5         Pessoa p1, p2, p3;  
6  
7         //Instanciação de tres objetos da classe Pessoa  
8         p1 = new Pessoa ("Flavia", 21);  
9         p2 = new Pessoa ("Carlos", 32);  
10        p3 = new Pessoa ("Bea", 40);  
11  
12        //Exibição dos nomes das pessoas  
13        System.out.println("Foram criados 3 objetos Pessoa com os nomes:");  
14        System.out.println(p1.getNome()+"."+p2.getNome()+"."+p3.getNome());  
15  
16        //Chamada do método para o Carlos fazer aniversário:  
17        p2.fazAniversario();  
18  
19        //Exibição dos dados do objeto após a execução do método fazAniversario  
20        System.out.println("Após fazer aniversário:");  
21        p2.exibeDados();  
22    }  
23}
```

6.5 Classe de teste com os dados lidos do teclado

```
1 public class TestePessoaComTeclado{
2     public static void main( String[] args) {
3         //Declaração de variáveis objeto
4         Pessoa p1, p2, p3;
5
6         //Instanciação de tres objetos da classe Pessoa
7         //lendo os dados do teclado
8         p1 = new Pessoa (Teclado.leString("\fDigite o nome da 1º pessoa: "),
9                         Teclado.leInt("Digite a idade: "));
10        p2 = new Pessoa (Teclado.leString("\nDigite o nome da 2º pessoa: "),
11                         Teclado.leInt("Digite a idade: "));
12        p3 = new Pessoa (Teclado.leString("\nDigite o nome da 3º pessoa: "),
13                         Teclado.leInt("Digite a idade: "));
14
15        //Exibição dos nomes das pessoas
16        System.out.println("\fForam criados 3 objetos Pessoa com os nomes:");
17        System.out.println(p1.getNome()+" , "+p2.getNome()+" e "+p3.getNome());
18
19        //Exibição dos dados da segunda pessoa antes de fazer aniversário:
20        System.out.println("\nSegunda pessoa, antes de fazer aniversário:");
21        p2.exibeDados();
22
23        //Chamada do método para a segunda pessoa fazer aniversário:
24        p2.fazAniversario();
25
26        //Exibição dos dados da segunda pessoa após a execução do método fazAniversario
27        System.out.println("\nSegunda pessoa,após fazer aniversário:");
28        p2.exibeDados();
29    } //fim do método main
30 } //fim da classe de teste
```

Figura 10 – Classe de teste, com entrada de dados.

Fonte: elaborada pela autora.

CAPÍTULO 7

A BIBLIOTECA JAVA

API é a denominação da biblioteca Java. Neste capítulo apresentaremos alguns pacotes e algumas classes dessa biblioteca.

7.1 Introdução

Uma biblioteca de software consiste em código já desenvolvido e devidamente depurado, pronto para ser usado. A plataforma Java oferece um conjunto imenso de funcionalidades organizadas em classes que, por sua vez, estão agrupadas em pacotes (*packages*), conforme a sua finalidade. Essa biblioteca é normalmente conhecida como *Application Programming Interface* (API) e figura como uma das fortes características da linguagem Java: o grande número de componentes reutilizáveis predefinidos.

Para especificar o uso de uma determinada classe de um pacote dentro de um programa, devemos incluir a declaração da importação do pacote, com a seguinte sintaxe:

```
import < aqui vai o nome do pacote>
```

A declaração acima permite que o programador instancie objetos e chame os métodos da(s) classe(s) importada(s).

Entre os pacotes oferecidos, listamos alguns com uma breve descrição da sua finalidade.

Quadro 7 – Alguns pacotes da biblioteca Java

| Pacote | Finalidade |
|-----------|--|
| java.io | Entrada e saída de dados. |
| java.lang | Classes fundamentais à programação Java. Entre muitas, <i>String</i> , <i>Math</i> e <i>Object</i> . |
| java.sql | Acesso a banco de dados. |
| java.text | Formatação de texto, datas e números. |
| java.util | Classes utilitárias. Entre muitas, <i>Scanner</i> . |

javax.swing | Interface gráfica com o usuário.

Fonte: elaborada pela autora.

7.2 O pacote Java.lang

O pacote Java.lang contém as classes básicas para a programação Java e não precisa ser explicitamente importado, pois é importado automaticamente.

Entre as classes que constam do pacote Java.lang, utilizaremos, com bastante frequência, alguns métodos das classes: Math, String e Object.

7.2.1 A classe Math

Essa classe contém métodos para realizar operações matemáticas elementares, tais como raiz quadrada, valor absoluto, exponenciação, logaritmo, funções trigonométricas e outras. Abaixo, uma relação de alguns métodos e constantes dessa classe.

Todos os métodos e constantes dessa classe são *static*, ou seja, pertencem à classe e não à instância. Isso significa que, para serem chamados, exigem o nome da classe e não uma variável-objeto.

Quadro 8 – Métodos

| Tipo de retorno | Método |
|------------------------|---|
| double | abs (double a) <i>Returns the absolute value of a double value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.</i> Exemplo: Math.abs(-3.5) resultado: 3.5 |
| int | abs (int a) |

| | |
|--------|---|
| | <p><i>Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.</i></p> <p>Exemplo: Math.abs(-3) resultado: 3</p> |
| double | <p>ceil (double a)</p> <p><i>Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.</i></p> <p>Exemplos: Math.ceil(4.2) resultado: 5.0 Math.ceil(4.8) resultado: 5.0</p> |
| double | <p>floor (double a)</p> <p><i>Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.</i></p> <p>Exemplos: Math.floor(4.2) resultado: 4.0 Math.floor(4.8) resultado: 4.0</p> |
| double | <p>pow (double a, double b)</p> <p><i>Returns the value of the first argument raised to the power of the second argument.</i></p> <p>Exemplos: Math.pow(3, 2) resultado: 9.0 Math.pow(-2, 3) resultado: -8.0</p> |
| double | <p>random ()</p> <p><i>Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.</i></p> <p>Exemplos: Math.random() resultado: 0.7171159550245632 Math.random() resultado: 0.036507852332339774</p> |

| | | | | | | | |
|--------------------------------|--|-------------------------------|----------------|--------------------------------|-----------------|-------------------------------|----------------|
| | <code>Math.random()</code> resultado: 0.9285823627588399 | | | | | | |
| double | <p><code>rint(double a)</code></p> <p><i>Returns the double value that is closest in value to the argument and is equal to a mathematical integer.</i></p> <p>Exemplos:</p> <table> <tr> <td><code>Math.rint(4.2)</code></td> <td>resultado: 4.0</td> </tr> <tr> <td><code>Math.rint(4.8)</code></td> <td>resultado: 5.0</td> </tr> <tr> <td><code>Math.rint(4.5)</code></td> <td>resultado: 4.0</td> </tr> </table> | <code>Math.rint(4.2)</code> | resultado: 4.0 | <code>Math.rint(4.8)</code> | resultado: 5.0 | <code>Math.rint(4.5)</code> | resultado: 4.0 |
| <code>Math.rint(4.2)</code> | resultado: 4.0 | | | | | | |
| <code>Math.rint(4.8)</code> | resultado: 5.0 | | | | | | |
| <code>Math.rint(4.5)</code> | resultado: 4.0 | | | | | | |
| long | <p><code>round (double a)</code></p> <p><i>Returns the closest long to the argument.</i></p> <p>Exemplos:</p> <table> <tr> <td><code>Math.round(4.8)</code></td> <td>resultado: 5</td> </tr> <tr> <td><code>Math.round(4.2)</code></td> <td>resultado: 4</td> </tr> <tr> <td><code>Math.round(4.5)</code></td> <td>resultado: 5</td> </tr> </table> | <code>Math.round(4.8)</code> | resultado: 5 | <code>Math.round(4.2)</code> | resultado: 4 | <code>Math.round(4.5)</code> | resultado: 5 |
| <code>Math.round(4.8)</code> | resultado: 5 | | | | | | |
| <code>Math.round(4.2)</code> | resultado: 4 | | | | | | |
| <code>Math.round(4.5)</code> | resultado: 5 | | | | | | |
| int | <p><code>round (float a)</code></p> <p><i>Returns the closest int to the argument.</i></p> <p>Exemplos:</p> <table> <tr> <td><code>Math.round(4.8f)</code></td> <td>resultado: 5</td> </tr> <tr> <td><code>Math.round(4.2f)</code></td> <td>resultado: 4</td> </tr> <tr> <td><code>Math.round(4.5f)</code></td> <td>resultado: 5</td> </tr> </table> | <code>Math.round(4.8f)</code> | resultado: 5 | <code>Math.round(4.2f)</code> | resultado: 4 | <code>Math.round(4.5f)</code> | resultado: 5 |
| <code>Math.round(4.8f)</code> | resultado: 5 | | | | | | |
| <code>Math.round(4.2f)</code> | resultado: 4 | | | | | | |
| <code>Math.round(4.5f)</code> | resultado: 5 | | | | | | |
| double | <p><code>signum (double d)</code></p> <p><i>Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.</i></p> <p>Exemplos:</p> <table> <tr> <td><code>Math.signum(4.8)</code></td> <td>resultado: 1.0</td> </tr> <tr> <td><code>Math.signum(-4.8)</code></td> <td>resultado: -1.0</td> </tr> <tr> <td><code>Math.signum(0)</code></td> <td>resultado: 0.0</td> </tr> </table> | <code>Math.signum(4.8)</code> | resultado: 1.0 | <code>Math.signum(-4.8)</code> | resultado: -1.0 | <code>Math.signum(0)</code> | resultado: 0.0 |
| <code>Math.signum(4.8)</code> | resultado: 1.0 | | | | | | |
| <code>Math.signum(-4.8)</code> | resultado: -1.0 | | | | | | |
| <code>Math.signum(0)</code> | resultado: 0.0 | | | | | | |
| float | <p><code>signum (float f)</code></p> <p><i>Returns the signum function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.</i></p> | | | | | | |

| | | | | | | | |
|--------------------|---|-------------------|----------------|--------------------|-----------------|-----------------|----------------|
| | <p>Exemplos:</p> <table> <tbody> <tr> <td>Math.signum(4.8f)</td><td>resultado: 1.0</td></tr> <tr> <td>Math.signum(-4.8f)</td><td>resultado: -1.0</td></tr> <tr> <td>Math.signum(0f)</td><td>resultado: 0.0</td></tr> </tbody> </table> | Math.signum(4.8f) | resultado: 1.0 | Math.signum(-4.8f) | resultado: -1.0 | Math.signum(0f) | resultado: 0.0 |
| Math.signum(4.8f) | resultado: 1.0 | | | | | | |
| Math.signum(-4.8f) | resultado: -1.0 | | | | | | |
| Math.signum(0f) | resultado: 0.0 | | | | | | |
| double | <p>sqrt (double a)</p> <p><i>Returns the correctly rounded positive square root of a double value.</i></p> <p>Exemplos:</p> <table> <tbody> <tr> <td>Math.sqrt(9)</td><td>resultado: 3.0</td></tr> <tr> <td>Math.sqrt(100)</td><td>resultado: 10.0</td></tr> </tbody> </table> | Math.sqrt(9) | resultado: 3.0 | Math.sqrt(100) | resultado: 10.0 | | |
| Math.sqrt(9) | resultado: 3.0 | | | | | | |
| Math.sqrt(100) | resultado: 10.0 | | | | | | |

Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

Quadro 9 – Constantes

| | | | |
|---------|---|---------|------------------------------|
| double | <p>E</p> <p><i>The double value that is closer than any other to e, the base of the natural logarithms.</i></p> <p>Exemplos:</p> <table> <tbody> <tr> <td>Math.E</td><td>resultado: 2.718281828459045</td></tr> </tbody> </table> | Math.E | resultado: 2.718281828459045 |
| Math.E | resultado: 2.718281828459045 | | |
| double | <p>PI</p> <p><i>The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.</i></p> <p>Exemplos:</p> <table> <tbody> <tr> <td>Math.PI</td><td>resultado: 3.141592653589793</td></tr> </tbody> </table> | Math.PI | resultado: 3.141592653589793 |
| Math.PI | resultado: 3.141592653589793 | | |

Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

7.2.2 A classe String

Essa classe representa as cadeias de caracteres, ou seja, uma sequência de caracteres (com índices a partir de zero). Strings são

constantes, seus valores não podem ser alterados após a criação.

A linguagem Java fornece o operador (+) para a concatenação de strings.

A seguir, uma relação de alguns métodos e constantes dessa classe. Alguns métodos dessa classe são *static* (essa palavra aparece junto ao tipo de retorno, indicando que o tal método é estático), ou seja, pertencem à classe e não à instância. Isso significa que, para serem chamados, exigem o nome da classe e não um objeto.

Quadro 10 – Alguns métodos da classe String

| Tipo de retorno e indicação de static | Método |
|---------------------------------------|---|
| char | charAt(int index) Returns the char value at the specified index. |
| int | compareTo(String anotherString) Compares two strings lexicographically. |
| int | compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences. |
| String | concat(String str) Concatenates the specified string to the end of this string. |
| boolean | contains(CharSequence s) Returns true if and only if this string contains the specified sequence of char values. |
| static String | copyValueOf(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified. |
| boolean | endsWith(String suffix) |

| | | |
|---------|---|---|
| | | Tests if this string ends with the specified suffix. |
| boolean | equals(Object anObject) | Compares this string to the specified object. |
| boolean | equalsIgnoreCase(String anotherString) | Compares this String to another String, ignoring case considerations. |
| void | getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) | Copies characters from this string into the destination character array. |
| int | indexOf(int ch) | Returns the index within this string of the first occurrence of the specified character. |
| int | indexOf(int ch, int fromIndex) | Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. |
| int | indexOf(String str) | Returns the index within this string of the first occurrence of the specified substring. |
| int | indexOf(String str, int fromIndex) | Returns the index within this string of the first occurrence of the specified substring, starting at the specified index. |
| int | lastIndexOf(int ch) | Returns the index within this string of the last occurrence of the specified character. |
| int | lastIndexOf(int ch, int fromIndex) | Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. |

| | |
|---------|--|
| int | lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring. |
| int | lastIndexOf(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index. |
| int | length() Returns the length of this string. |
| String | replace(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar. |
| boolean | startsWith(String prefix) Tests if this string starts with the specified prefix. |
| boolean | startsWith(String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index. |
| String | substring(int beginIndex) Returns a new string that is a substring of this string. |
| String | substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string. |
| char[] | toCharArray() Converts this string to a new character array. |
| String | toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale. |
| String | toUpperCase() |

| | |
|---------------|--|
| | Converts all of the characters in this String to upper case using the rules of the default locale. |
| String | trim() Returns a copy of the string, with leading and trailing whitespace omitted. |
| static String | valueOf(boolean b) Returns the string representation of the boolean argument. |
| static String | valueOf(char c) Returns the string representation of the char argument. |
| static String | valueOf(char[] data) Returns the string representation of the char array argument. |
| static String | valueOf(char[] data, int offset, int count) Returns the string representation of a specific subarray of the char array argument. |
| static String | valueOf(double d) Returns the string representation of the double argument. |
| static String | valueOf(float f) Returns the string representation of the float argument. |
| static String | valueOf(int i) Returns the string representation of the int argument. |
| static String | valueOf(long l) Returns the string representation of the long argument. |
| static String | valueOf(Object obj) Returns the string representation of the Object argument. |

7.2.3 A classe Object

Em Java, todas as classes são, automaticamente, subclasses da classe Object, a qual é a raiz da hierarquia de classes. Ela oferece alguns métodos bem gerais que podem ser sobreescritos. Os principais métodos da classe Object são listados abaixo:

Quadro 11 – Alguns métodos da classe Object

| Visibilidade e retorno | Método |
|-------------------------------|---|
| protected Object | <u>clone</u> () Creates and returns a copy of this object. |
| boolean | <u>equals</u> (Object obj) Indicates whether some other object is "equal to" this one. |
| Class<? extends Object> | <u>getClass</u> () Returns the runtime class of an object. |
| String | <u>toString</u> () Returns a string representation of the object. |

Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

7.3 O pacote Java.util

Esse pacote contém uma miscelânea de classes utilitárias, as coleções e a classe que usaremos para capturar valores do teclado. Entre as classes que constam do pacote Java.util utilizaremos, com bastante frequência, alguns métodos da classe Scanner.

7.3.1 A classe Scanner

Quadro 12 – Alguns métodos da classe Scanner

| tipo de retorno | Método |
|------------------------|---|
| byte | <u>nextByte</u> () Scans the next token of the input as a byte. |
| byte | <u>nextByte</u> (int radix) Scans the next token of the input as a byte. |
| double | <u>nextDouble</u> () Scans the next token of the input as a double. |
| float | <u>nextFloat</u> () Scans the next token of the input as a float. |
| int | <u>nextInt</u> () Scans the next token of the input as an int. |
| int | <u>nextInt</u> (int radix) Scans the next token of the input as an int. |
| String | <u>nextLine</u> () Advances this scanner past the current line and returns the input that was skipped. |
| long | <u>nextLong</u> () Scans the next token of the input as a long. |
| long | <u>nextLong</u> (int radix) Scans the next token of the input as a long. |
| short | <u>nextShort</u> () Scans the next token of the input as a short. |
| short | <u>nextShort</u> (int radix) |

Scans the next token of the input as a short.

Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

Resumindo:

`nextInt();` para ler um valor do tipo int
`nextFloat();` para ler um valor do tipo float
`nextDouble();` para ler um valor do tipo double
`nextChar();` para ler um valor do tipo char
`nextLine();` para ler um valor do tipo String

7.3.2 Como utilizar a classe Scanner

Para utilizá-la é necessário importar a classe:

```
import java.util.Scanner;
```

Depois de importar a classe, devemos instanciar um objeto da classe Scanner:

```
Scanner leia = new Scanner(System.in);
```

Obs.: *leia* é um nome de variável que foi escolhido porque essa variável será usada para chamar os métodos de leitura. Obviamente, poderia ser escolhido qualquer outro nome. Por exemplo: *entrada*, *teclado*, *read* etc.

Após a instanciação, poderemos ler valores através do teclado:

- Lendo um valor inteiro: `int n = leia.nextInt();`
- Lendo um valor real (float): `float preço = leia.nextFloat();`

- Lendo um valor real (double): **double** salário = leia.nextDouble();
- Lendo uma String: **String** palavra = leia.nextLine();

7.3.3 Exemplo

O exemplo a seguir contém entrada de dados pelo teclado, processamento com o uso do comando de atribuição e saída para a tela. Suponha que alguém deseja um programa que seja capaz de calcular a média aritmética de duas notas obtidas por um aluno e de exibir a média calculada na tela.

Devemos, então, desenvolver um programa que receba as notas do aluno pelo teclado, calcule a média aritmética e exiba a média calculada na tela.

Desenvolver um programa consiste em escrever as instruções que o computador deve executar para realizar a tarefa especificada. Independentemente da linguagem de programação, a lógica, isso é, a sequência de instruções poderia ser a seguinte:

- 1º) *solicite que uma nota seja digitada, exibindo uma mensagem na tela;*
- 2º) *leia a nota do teclado e armazene em uma variável na memória;*
- 3º) *solicite que a outra nota seja digitada, exibindo uma mensagem na tela;*
- 4º) *leia a outra nota do teclado e armazene em uma variável na memória;*
- 5º) *calcule a média aritmética e armazene em uma variável na memória;*
- 6º) *exiba na tela o valor da média.*

A sequência de instruções, em Java:

```
// instancia um objeto do tipo Scanner  
Scanner leia = new Scanner(System.in);  
  
// declara as variáveis  
double n1, n2;  
double media;  
  
// lê as duas notas do teclado  
System.out.println ("Digite uma nota:");  
n1 = leia.nextDouble();  
System.out.println ("Digite outra nota:");  
n2 = leia.nextDouble();  
  
// calcula a média  
media = (n1 + n2)/2;  
  
// exibe a média na tela  
System.out.printf ("Media: %4.2f",media);
```

7.4 Documentação completa da API

A Oracle disponibiliza a documentação da API no endereço <http://docs.oracle.com/javase/7/docs/api/> para consulta ou para ser baixada para acesso local. Os programadores podem navegar nessa documentação, clicando nos itens que aparecem nos quadros, conforme a figura.

The screenshot shows a browser window displaying the Java API documentation for the `Math` class. The URL is `docs.oracle.com/javase/7/docs/api/`. The left sidebar lists various Java packages and interfaces. The main content area shows the `Math` class definition:

```

public final class Math
extends Object

```

The class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. Unlike some of the numeric methods of class `StrictMath`, all implementations of the equivalent functions of class `Math` are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the `Math` methods simply call the equivalent method in `StrictMath` for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where appropriate, to provide higher-performance implementations of `Math` methods. Such higher-performance implementations still must conform to the specification for `Math`.

The quality of implementation specifications concern two properties, accuracy of the returned result and monotonicity of the method. Accuracy of the floating-point `Math` methods is measured in terms of *ups*, units in the last place. For a given floating-point format, an *up* of a specific real number b is the distance between the two floating-point values closest to b meeting that requirement. If the argument has an exact representation as a value of an open interval of *ups*, the *up* nearest to the exact result is for the worst-case error at any argument. If a method always has an error less than 0.5 *ups*, it always returns the floating-point number nearest the exact result, such a method is correctly rounded. A correctly rounded method is generally the best a floating-point approximation can be; however, it is impractical for many floating-point methods to be correctly rounded. Instead, for the `Math` class, a larger error bound of 1 or 2 *ups* is allowed for certain methods. Informally, with a 1 *up* error bound, when the exact result is a multiple of a *up*, the method should be returning the closest floating-point value, otherwise, it should be returning a floating-point value which bracketed the exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be refined. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 *up* errors are required to be semi-monotonic: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 *up* accuracy will automatically meet the monotonicity requirements.

Since: JDK1.0

Field Summary

| Fields | |
|--------------------------|--|
| Modifier and Type | Field and Description |
| static double | E The double value that is closer than any other to e, the base of the natural logarithms. |
| static double | PI The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter. |

Method Summary

| Methods | |
|--------------------------|--|
| Modifier and Type | Method and Description |
| static double | <code>abs(double a)</code> Returns the absolute value of a <code>double</code> value. |
| static float | <code>abs(float a)</code> Returns the absolute value of a <code>float</code> value. |

Figura 11 – API.

Fonte: <http://docs.oracle.com/javase/7/docs/api/>.

7.5 Exemplos de uso de alguns métodos das classes Math e String

```
public class AlgunsExemplosDeMetodosDasClassesJava{
    public static void main( String[] args)      {
        int num = 9;
        int x = 7421;
        String nome = "java";
        char letra1 ,letra2,letra3 ,letra4;

        System.out.println ("\fnum = " + num );
        System.out.println ("Raiz quadrada de num: " + Math.sqrt(num));
        System.out.println ("Quadrado de num: " + Math.pow(num,2));

        //converte inteiro para String
        String stx = String.valueOf(x);
        //pega o tamanho do String
        int tamanho = stx.length();
        System.out.println ("\nO número "+ x + " tem "+ tamanho + " dígitos\n");

        //Exibe o nome ocupando 19 colunas, ajustado para a direita
        System.out.printf("%19s\n", nome);
        nome = nome.toUpperCase(); //Converte em letras maiúsculas
        letra1= nome.charAt(0); //pega a primeira letra
        letra2= nome.charAt(1); //pega a segunda letra
        letra3= nome.charAt(2); //pega a terceira letra
        letra4= nome.charAt(3); //pega a quarta letra
        //Exibe as letras do nome, ocupando 5 colunas para cada letra
        System.out.printf("%5c%5c%5c%5c\n\n",letra1,letra2,letra3,letra4);
        //Compara Strings
        System.out.println("JAVA eh igual a Java: " + nome.equals("Java") );
    }
}
```

Tela de saída:

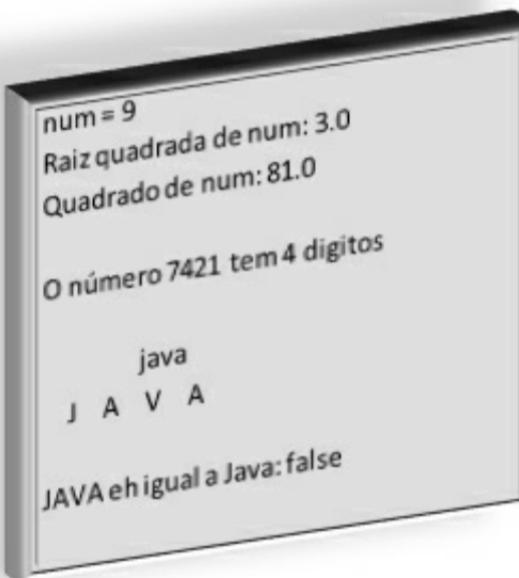


Figura 12 – Exemplos de chamada de alguns métodos das classes String e Math.
Fonte: elaborada pela autora.

CAPÍTULO 8

ESTRUTURAS DE SELEÇÃO

Neste capítulo veremos as estruturas de seleção ou comandos de decisão. A linguagem Java oferece dois comandos de seleção: *if* e *switch case*. Apresentaremos as várias sintaxes, com exemplos.

8.1 Introdução

Na maioria das linguagens de programação existem as chamadas estruturas de decisão ou instruções condicionais. Essas estruturas são comandos que permitem desviar o fluxo de controle do programa, com base em uma condição. Esse tipo de comando é conhecido como comando “se” ou comando “*if*”, ou ainda, como comando “se então” ou “*if else*”.

Outra utilização dos comandos de seleção é a necessidade de verificar opções de escolha. Por exemplo, o programa exibe um *menu* e solicita que o usuário escolha uma das opções. O programa capta a opção escolhida em uma variável e precisa verificar qual é a opção desejada pelo usuário para transferir o controle da execução para a parte do programa adequada. Embora isso possa ser feito com um comando “*if*”, há um outro comando mais apropriado para ser aplicado nesses casos: é o comando conhecido como “caso” ou “*case*”, ou ainda, “*switch case*”, em Java.

8.2 Estrutura ou comando de seleção (*if*)

Usaremos esses comandos quando, no decorrer do programa, quisermos especificar que certos comandos devem ser ou não executados conforme certas condições. As condições são expressas através de uma expressão booleana. Podemos estabelecer essa execução condicional para um único comando ou para um conjunto de comandos delimitado por chaves (“{”e “}”), chamado bloco de comandos.

8.2.1 Sintaxes

```
if (expressão booleana)  
    comando;
```

Semântica

Se a condição for verdadeira, o comando será executado; se não for, o fluxo de execução segue após o comando sem executá-lo.

```
if (expressão booleana) {  
  
    comando_1;  
    comando_2;  
    .....  
  
}
```

Semântica

Se a condição for verdadeira, o bloco de comandos será executado; se não for, o fluxo de execução segue após o bloco, sem executá-lo.

```
if (expressão booleana)  
    comando_a;  
else  
    comando_b;
```

Semântica

Se a condição for verdadeira, somente o comando_a será executado; se não for, somente o comando_b será executado.

```
if (expressão_booleana) {  
  
    comando_a;  
    comando_b;  
    .....  
  
}  
  
else {  
  
    comando_x;  
    comando_y;  
    .....  
  
}
```

Semântica

Se a condição for verdadeira, somente o primeiro bloco (comando_a, comando_b...) será executado; se não for, somente o segundo bloco (comando_x, comando_y) será executado.

8.2.2 Exemplos

- Em determinado ponto de um programa, o programador quer “dizer” ao computador que execute o comando que calcula a raiz quadrada de um número, somente se o numero for positivo.

Em português:

```
se (numero >0) então  
raiz = √ numero
```

Em Java:

```
if (numero > 0)  
raiz = Math.sqrt (numero);
```

- Em determinado ponto de um programa, o programador quer “dizer” ao computador que execute o comando que exibe na tela a mensagem “Aprovado”, somente se a média do aluno for maior ou igual a 5,0. Do contrário, o programador quer que o computador execute outro comando, ou seja, o que exibe na tela a mensagem “Reprovado”.

Em português:

```
se (média ≥ 5,0) então
    escreva “Aprovado”
senão escreva “Reprovado”
```

Em Java:

```
if (media >= 5.0)
    System.out.println ("Aprovado");
else
    System.out.println "Reprovado";
```

- Em determinado ponto de um programa, o programador quer “dizer” ao computador que execute o comando que exibe na tela a mensagem “Ótimo” se a média for maior ou igual a 9,0. Do contrário, o programador quer que o computador execute outro teste: verifique se a média é maior ou igual a 8,0 e, nesse caso, exiba na tela a mensagem “Bom”. Do contrário, o programador quer que o computador execute ainda outro teste: verifique se a média é maior ou igual a 6,0 e, nesse caso, exiba na tela a mensagem “Aprovado”. Do contrário, exiba na tela a mensagem “Reprovado”.

Em português:

```
se (média >= 9.0) então
    escreva (“Ótimo”)
senão se (média >= 8.0) então
    escreva (“Bom”)
senão se (média >= 6.0) então
    escreva (“Aprovado”)
senão
    escreva (“Reprovado”)
```

Em Java:

```
if (media >= 9.0)
    System.out.println ("Otimo");
else if (media >= 8.0)
    System.out.println ("Bom");
else if (media >= 6.0)
    System.out.println ("Aprovado");
else
    System.out.println ("Reprovado");
```

8.3 Estrutura de seleção case (switch)

Utilizaremos esse comando quando quisermos selecionar um entre

vários comandos ou blocos de comandos mediante a comparação de uma variável do tipo byte, char, short ou int com valores constantes.

8.3.1 Sintaxe

```
switch (variável) {  
    case valor 1 : comando_a;  
                    comando_b;  
                    ...  
                    break;  
  
    case valor 2 : comando_x;  
                    comando_y;  
                    ...  
                    break;  
    ...  
    case valor N : comando_m;  
                    comando_n;  
                    ...  
                    break;  
  
    [default :bloco_de_comandos;]  
}
```

Semântica

O valor da variável é comparado com valor 1. Se igual, executa os comandos que fazem parte deste bloco até atingir o comando break, quando então segue o fluxo de execução após o switch. Se não é igual, a variável é comparada com valor 2, valor 3 e assim sucessivamente. Se o valor da variável não for igual a nenhum dos valores especificados, então executa o bloco de comandos que estão em default.

Obs.: O comando break normalmente é utilizado como último comando de cada bloco, indicando que, quando um bloco for executado, todos os outros devem ser ignorados. Os colchetes [] indicam que essa parte do comando é opcional.

8.3.2 Exemplos

- Suponha que um programa use um código de 1 a 7 (domingo a sábado) para armazenar o dia da semana em uma variável do tipo inteira. Em determinado ponto do programa, o programador quer

“dizer” ao computador para escrever na tela o nome do dia da semana, ou “Fim de semana”, se for sábado ou domingo, correspondente ao número armazenado na variável dia.

Em português:

Valor da variável (dia)

caso seja 2: escreva (“Segunda-feira”)

saia

caso seja 3: escreva (“Terça-feira”)

saia

caso seja 4: escreva (“Quarta-feira”)

saia

caso seja 5: escreva (“Quinta-feira”)

saia

caso seja 6: escreva (“Sexta-feira”)

saia

nenhum dos acima: escreva (“Fim de semana”)

Em Java:

```
switch (dia)
case 2: System.out.println ("Segunda-feira");
           break;
case 3: System.out.println ("Terça-feira");
           break;
case 4: System.out.println ("Quarta-feira");
           break;
case 5: System.out.println ("Quinta-feira");
           break;
case 6: System.out.println ("Sexta-feira");
           break;
default: System.out.println ("Fim de semana");
```

8.4. Métodos utilizando comandos de seleção

- Método que recebe um número inteiro e exibe na tela a mensagem “Negativo” ou “Não negativo”, conforme o caso.

```
public void exibeMensagem(int num){
    if (num < 0)
        System.out.println ("Negativo");
    else
        System.out.println ("Não negativo");
}
```

- Método que recebe um número inteiro e retorna a mensagem “Par” ou “Impar”, conforme o caso.

```
public String verificaParOuImpar(int num){  
    if (num%2==0)  
        return "Par";  
    else  
        return "Impar"; //aqui else é opcional  
}
```

- Método que recebe dois números reais e retorna o maior. Se iguais, retorna qualquer um.

```
public double maiorDosDois(double a, double b){  
    if (a>b)  
        return a;  
    else  
        return b; //aqui else é opcional  
}
```

- Método que recebe dois números inteiros e retorna uma das seguintes mensagens, conforme o caso: “Os dois são diferentes”, ou “Os dois são iguais”.

```
public String comparaNumeros(int umNum, int outroNum){  
    if (umNum != outroNum)  
        return ("Os dois são diferentes");  
    else  
        return ("Os dois são iguais");  
}
```

- Método que recebe dois nomes de pessoas, do tipo String, e retorna true se os dois nomes são iguais e false se são diferentes. Observe que a comparação de strings é realizada com os métodos da classe String, pois String é um objeto.

```
public boolean iguais(String umNome, String outroNome){  
    if (umNome.equalsIgnoreCase (outroNome))  
        return true;  
    else          //aqui else é opcional  
        return false;  
}
```

- Método que recebe a idade de uma pessoa, e classifica-a da seguinte forma: até 12 anos, “Infantil”; de 13 a 18, “Adolescente”; de 19 a 24, “Jovem”; de 25 a 70, “Adulto”; mais de 70 anos, “Terceira idade”.

```
public String faixaEtaria (int idade){  
    String msg = “Terceira idade”;  
    if (idade<= 12 )  
        msg = “Infantil”;  
    else  if (idade<= 18)  
        msg = “Adolescente”;  
    else  if (idade< 25 )  
        msg = “Jovem”;  
    else  if (idade<= 70 )  
        msg = “Adulto”;  
    return msg;  
}
```

- ♦ Método que recebe o ano e retorna true ou false, indicando se o ano é bissexto ou não.

```
public boolean ehBissexto(int ano) {  
    if ((ano % 4 == 0)  
        &&(ano % 100 != 0)  
        ||(ano % 400 == 0))  
        return true;  
    else return false; //aqui else é opcional  
}
```

- Método que recebe o ano e o mês e retorna o número de dias do mês.

```
public int diasDoMes(int ano, int mes){  
    int dias;  
    switch (mes) {  
        case 4 :  
        case 6 :  
        case 9 :  
        case 11 :    dias = 30;  
                      break;  
  
        case 1 :  
        case 3 :  
        case 5 :  
        case 7 :  
        case 8 :  
        case 10 :  
        case 12 :    dias = 31;  
                      break;  
  
        case 2 :    dias = 28;  
                     if (ehBissexto (ano))  
                         dias = 29;  
    }  
    return dias;  
}
```

CAPÍTULO 9

ESTRUTURAS DE REPETIÇÃO

Neste capítulo veremos as estruturas ou comandos de repetição ou iteração. Apresentamos a sintaxe, a semântica e exemplos dessas estruturas. A linguagem Java oferece três comandos de iteração, a saber, *while*, *do-while* e *for*. Esses comandos permitem repetir a execução de acordo com certas condições, estabelecidas pelo programador.

9.1 Introdução

Na maioria das linguagens de programação existem as chamadas estruturas de repetição ou comandos de iteração ou laços. Essas estruturas são comandos que permitem repetir a execução de uma parte do programa um número específico de vezes, ou até que uma determinada condição torne-se verdadeira ou falsa. Esse tipo de comando é conhecido como comando “enquanto” ou comando “*while*”, ou ainda, como comando “para” ou “*for*”.

As estruturas de repetição disponíveis em Java são: *while*, *for* e *do-while*.

9.2 Estrutura while

Esse é o comando “Enquanto a expressão for verdadeira, faça o(s) comando(s)”.

9.2.1 Sintaxes

```
while (expressão booleana)  
comando;
```

Esta sintaxe é utilizada para repetir a execução de apenas um comando.

```
while (expressão booleana) {  
comando_1;  
comando_2;  
comando_3;  
.....  
}
```

Esta sintaxe é utilizada para repetir a execução de um bloco de comandos (mais de um).

9.2.2 Semântica

Nessa estrutura, enquanto a condição for verdadeira, o código dentro do laço é executado. Quando a condição tornar-se falsa, o primeiro comando, após o final do laço, será executado. Nesse caso, se a condição for falsa já na primeira vez, o laço não será executado nenhuma vez.

É importante notar que os comandos dentro do laço devem ser capazes de alterarem a condição para que essa se torne falsa, pois, caso contrário, a condição será sempre verdadeira, e os comandos dentro do laço ficarão em execução infinitamente, ou seja, o programa fica em *loop*.

9.2.3 Exemplo

Suponha que, em determinado ponto de um programa, seja necessário que se exiba, na tela, os números inteiros de 1 a 10.

Para que essa tarefa seja realizada, além de especificar o comando que gera um número inteiro e o comando que exibe o número gerado, o programador deve “dizer” ao computador que execute esses comandos dez vezes (ou enquanto o número gerado e exibido estiver entre 1 e 10).

Em português:

```
num = 0
enquanto (num <10) faça
    incrementa num
    escreva num
```

Em Java:

```
num = 0
while (num<10) {
    num = num + 1;
    System.out.println (num);
}
```

9.3 Estrutura for

Esse é o comando “*Faça o(s) comando(s) enquanto a variável vai de valor inicial até o valor final, atribuindo um novo valor à variável a cada iteração*”.

9.3.1 Sintaxes

A sintaxe abaixo é usada para repetir um comando.

```
for (inicializa variável; expressão booleana; atribui_novo_valor_à_variável)
    comando;
```

A sintaxe abaixo é usada para repetir um bloco de comandos (mais de um).

```
for (inicializa variável; expressão booleana; atribui_novo_valor_à_variável) {  
    comando_1;  
    comando_2;  
    comando_3;  
    .....  
}
```

9.3.2 Semântica

Essa estrutura cria um laço de repetição no fluxo do programa, baseado em três partes:

- ***inicializa variável*** – É a expressão inicial que é executada apenas uma vez, na entrada do laço. Atribui o valor inicial para a variável de controle do laço. A variável pode ser declarada junto com a inicialização.
- ***expressão booleana*** - É a condição que é avaliada a cada iteração do laço e determina quando a repetição deve ser abandonada. Caso a condição seja verdadeira, executam-se o(s) comando(s) do laço, caso seja falsa, o controle pula para a próxima instrução, seguinte ao laço.
- ***atribui novo valor à variável*** - É uma atribuição, executada a cada iteração. Geralmente é usada para incrementar ou decrementar a variável de controle do laço.

9.3.3 Exemplo

Vamos usar o mesmo exemplo anterior: exibir, na tela, os números

inteiros de 1 a 10. Observe que, com apenas um comando (o comando *for*), inicializamos a variável, incrementamos a variável e testamos a condição.

Em português:

para num começando em 1, até 10, de 1 em 1 faça

escreva num

Em Java:

```
for (int num= 1; num<=10; num++) {  
    System.out.println (num);  
}
```

9.4 Estrutura do-while

Esse é o comando “faça o(s) comando(s) enquanto a expressão for verdadeira”.

9.4.1 Sintaxes

```
do  
  comando;  
while (expressão booleana);
```

A sintaxe abaixo é usada para repetir apenas um comando.

```
do {  
  comando_1;  
  comando_2;  
  comando_3;  
  .....  
}  
while (expressão booleana);
```

A sintaxe abaixo é usada para repetir um bloco de comandos (mais de um).

9.4.2 Semântica

Nessa estrutura, o código dentro do laço será executado pelo menos uma vez, pois o teste somente é feito no final, e continuará sendo executado enquanto a condição for falsa. Quando a condição tornar-se verdadeira, o primeiro comando, após o final do laço, será executado. Nesse caso, os comandos são executados pelo menos uma vez.

É importante notar que os comandos dentro do laço devem ser capazes de alterarem a condição para que essa se torne verdadeira, pois, caso contrário, a condição será sempre falsa, e os comandos dentro do laço ficarão em execução infinitamente, ou seja, o programa fica em *loop*.

9.4.3 Exemplo

Vamos usar o mesmo exemplo anterior: exibir, na tela, os números inteiros de 1 a 10.

Em português:

```
num = 1  
faça  
    escreva num  
    incrementa num  
enquanto num <= 10
```

Em Java:

```
num = 1;  
do  
{  
    System.out.println (num);  
    num = num +1;  
}  
while (num <= 10);
```

9.5 Métodos que utilizam estruturas de repetição

- Método que exibe na tela os números inteiros de 20 a 1, lado a lado.

```
public void exibeInteiros() {  
    for (int num = 20; num>0; num--)  
        System.out.print(num + " ");  
}
```

- Método que recebe, via parâmetros, dois números inteiros (x e y) e exibe os quadrados dos números inteiros de x até y.

```
public void exibeQuadrados(int x, int y){  
    for (int num = x; num<=y; num++)  
        System.out.println(Math.pow(num,2));  
}
```

- Método que recebe, via parâmetros, dois números inteiros (x e y) e exibe os quadrados dos números inteiros pares e a raiz quadrada dos ímpares, de x até y.

```
public void exibeQuadradosEraiz(int x, int y){  
    for (int num = x; num<=y; num++)  
        if (num%2==0)  
            System.out.println("quadrado: "+Math.pow(num,2));  
        else  
            System.out.println("raiz quad.: "+Math.sqrt(num));  
}
```

- ♦ Método que recebe, via parâmetro, um número inteiro (n) e gera e exibe na tela n números aleatórios inteiros, usando o método Math.random. O método calcula e retorna a soma dos números gerados.

```
public int somaNumerosGerados(int n){  
    int soma = 0;  
    int num;  
    for (int i = 1; i<=n; i++){  
        num = (int)( Math.random( )*100);  
        System.out.println(num);  
        soma = soma +num;  
    }  
    return soma;  
}
```

- ♦ Método que exibe na tela as letras maiúsculas de 'A' a 'Z'.

```
public void exibeLetrasMaiusculas( ){  
    for (char letra ='A'; letra <='Z'; letra++)  
        System.out.println(letra+ " " );  
}
```

- Método que recebe um inteiro n e desenha na tela um triângulo de n linhas na forma exemplificada para n=5.

55555

4444

333

22

1

```
public void desenhaTriangulo (int n ) {  
    for (int i=n; i>=1;i--) {  
        for (int j=1; j<=i;j++) {  
            System.out.print(i);  
        }  
        System.out.println();  
    }  
}
```

- Método que lê do teclado e valida a resposta do usuário. Só aceita 'S' ou 'N'. O método retorna a resposta.

```
public char leResposta(){  
    char resp;  
    do {  
        resp = Teclado.leChar("S - Sim ou N - não ?");  
        resp = Character.toUpperCase(resp); //converte para maiúscula  
    }  
    while (resp != 'S'&& resp != 'N');  
    return resp;  
}
```

- Método que gera e exibe na tela números aleatórios inteiros, no intervalo [1,10], usando o método Math.random. O método deve encerrar quando a soma dos números gerados ultrapassar 25.

```
public void geraNumerosExibe( ){  
    int soma = 0;  
    int num;  
    while (soma <= 25){  
        num = 1+ (int)( Math.random( )*10);  
        System.out.println(num);  
        soma = soma +num;  
    }  
}
```

- Método que recebe um inteiro **n** e desenha na tela um triângulo de **n** linhas na forma exemplificada para **n=5**.

```
1  
22  
333  
4444  
55555
```

```
public void desenhaTriangulo (int n ) {  
    for (int i=1; i<=n; i++){  
        for (int j=1; j<=i; j++)  
            System.out.print(i);  
        System.out.println();  
    }  
}
```

- Método que lê do teclado uma série de notas dos alunos de uma turma. Encerra a leitura no momento em que ler a nota fictícia -1. Retorna a média da turma. Só aceita notas de 0.0 a 10.0.

```
public double calculaMediaDaTurma(){  
    double n; //variável que armazena cada nota lida do teclado  
    int cont = 0; //contador de notas  
    double soma = 0; //acumulador de notas  
  
    do {  
        n = Teclado.leDouble("nota[-1 p/ encerrar]:");  
        if (n<0 || n>10 && n!= -1)  
            System.out.println("nota inválida");  
    }  
  
    while ((n<0 || n>10) && n!= -1);  
    while (n != -1){  
  
        soma = soma + n;  
        cont++;  
  
        do {  
            n = Teclado.leDouble("nota[-1 p/ encerrar]:");  
            if (n<0 || n>10 && n!= -1)  
                System.out.println ("nota inválida");  
        }  
  
        while ((n<0 || n>10) && n!= -1);  
    }  
  
    if(cont>0)  
        return soma/cont;  
    else return 0;
```

Obs. Observe a repetição de código. No próximo exemplo é eliminado o código

duplicado.

- ♦ Método privado (para uso exclusivo dos outros métodos da própria classe) que lê uma nota do teclado e retorna a nota lida. Só aceita notas de 0 a 10 (com exceção de -1).

```
private double leUmaNota () {  
    double nota;  
    do{  
        nota = Teclado.leDouble("nota[-1 p/ encerrar]:");  
        if (nota<0 || nota>10 && nota!= -1)  
            System.out.println("nota inválida");  
    } while ((nota<0 || nota>10) && nota!= -1);  
    return nota;  
}
```

- Método que lê do teclado uma série de notas dos alunos de uma turma. Encerra a leitura no momento em que ler a nota fictícia -1. Retorna a média da turma. Só aceita notas de 0 a 10. Obs.: para evitar código duplicado, a leitura é feita chamando o método *leUmaNota*.

```
public double calculaMediaDaTurma(){  
    double n;  
    int cont = 0;//conta as notas  
    double soma =0;//acumula as notas  
    n = leUmaNota();  
    while (n != -1){  
        soma = soma + n;  
        cont++;  
        n = leUmaNota();  
    }  
    if(cont>0)  
        return soma/cont;  
    else return 0;  
}
```

CAPÍTULO 10

HERANÇA E POLIMORFISMO

Este capítulo trata do mecanismo de herança, que permite a especialização e generalização de classes, com aproveitamento de código. É um conceito obrigatoriamente presente no paradigma orientação a objetos, independentemente da linguagem de programação. Além de analisar o conceito, veremos como esse recurso é implementado em Java.

10.1 Introdução

Frequentemente, precisamos projetar classes que possuem comportamento e características comuns. Em vez de desenvolvermos cada classe de forma independente e, com isso, duplicarmos código, podemos usar o recurso de herança e projetar uma hierarquia de classes em que a superclasse (a classe que figura no topo da hierarquia) defina a parte comum, e cada subclasse herde essa parte comum e acrescente o comportamento e o conhecimento específicos a sua natureza.

A herança também nos permite especializar uma classe já existente (da API ou da própria organização). Ao desenvolvermos uma nova classe, devemos fazer a pergunta “é um(a)?” em relação aos objetos das classes já existentes. Se a resposta for sim, podemos estender a classe existente.

10.2 O que é herança

É uma forma de reutilização de software que permite estender (herdar) uma classe para criar outras mais especializadas. Uma classe só pode estender (herdar) uma única classe. Mas várias classes podem estender (herdar) a mesma classe.

A hierarquia de classes, em forma de árvore, parte de uma classe mais genérica para as classes mais especializadas.

A classe mais geral é chamada de superclasse. A classe mais especializada (que herda da superclasse) é chamada de subclasse. Outras denominações comuns para a superclasse são *classe-pai* e *classe-base* e, para a subclasse, são *classe-filha* e *classe derivada*.

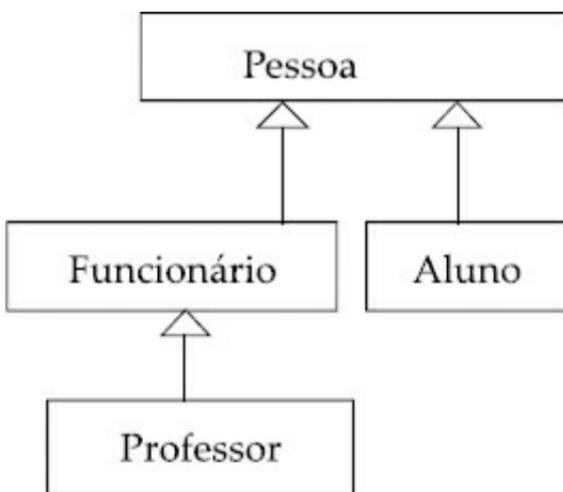
Herdar, estender ou especializar uma classe consiste em desenvolver uma nova classe, que aproveita todo o conhecimento (as

características) da superclasse e, ainda, pode definir novas características. Quanto ao comportamento, aproveita o que é adequado, modifica o que não serve e acrescenta o que for necessário.

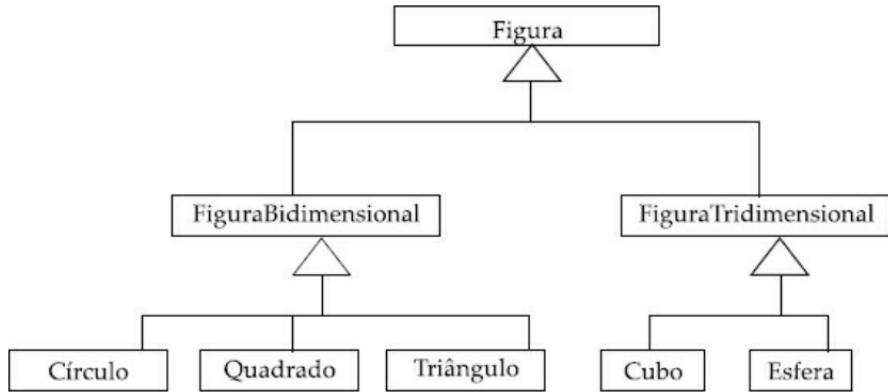
Em Java, cada classe que não estende especificamente outra classe é automaticamente uma subclasse da classe *Object*. Então, cada classe estende a classe *Object* (do pacote `java.lang`), direta ou indiretamente.

Os relacionamentos de herança entre as classes podem ser visualizados usando as setas, em UML. Considere a hierarquia abaixo e observe que cada seta representa um relacionamento é-um. Seguindo as setas, podemos dizer que professor é um funcionário, professor é uma pessoa, funcionário é uma pessoa e aluno é uma pessoa.

A superclasse *Pessoa* tem duas subclasses: *Funcionário* e *Aluno*. *Funcionário* também é uma superclasse que tem como subclasses *Professor*.

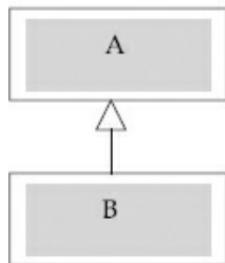


O mesmo ocorre na hierarquia abaixo, em que também podemos aplicar o teste é-um(a): cubo é uma figura ou cubo é uma figura tridimensional, quadrado é uma figura ou quadrado é uma figura bidimensional. Círculo, Quadrado e Triângulo são subclasses de *FiguraBidimensional* que, por sua vez, é uma subclasse de *Figura*. A superclasse *Figura* tem duas subclasses: *FiguraBidimensional* e *FiguraTridimensional*.



10.2.1 Como especificar herança

Em UML



Em Java

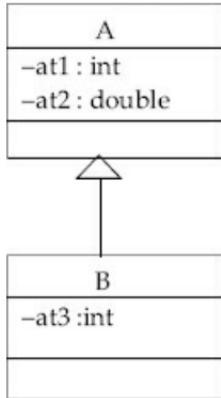
```
public class B extends A
```

Figura 13 – Relacionamento de herança em UML e em Java.
Fonte: elaborada pela autora.

10.2.2 O que acontece com os atributos

Ao usar herança, todos os atributos da superclasse são herdados pela subclasse, e novos atributos podem ser declarados para a subclasse. Observe que os atributos herdados não são mencionados na subclasse.

Em UML



Em Java

```
public class B extends A {  
    private int at3; // declara mais um atributo
```

Os objetos da subclasse B têm três atributos. Exemplo:

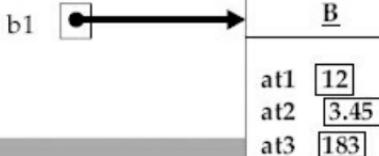


Figura 14 – Como herdar atributos da superclasse e declarar novos atributos.
Fonte: elaborada pela autora.

10.2.3 O que acontece com os construtores

Ao usar herança, os construtores da superclasse não são herdados pela subclasse. A subclasse deve oferecer seus próprios construtores. Mas aqueles atributos que foram herdados devem ser configurados pelo construtor da superclasse. Então, o construtor da subclasse deve chamar o construtor da superclasse. Esse comando de chamada deve ser o primeiro comando do construtor da subclasse.

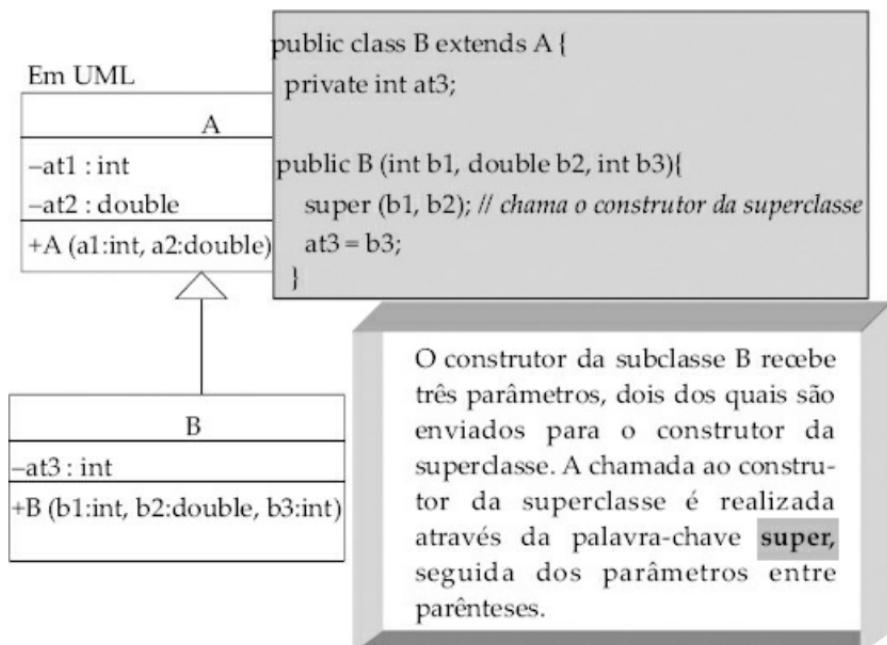
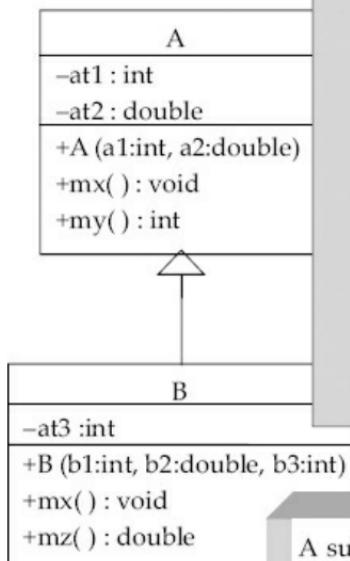


Figura 15 – Construtores da super e da subclasse
Fonte: elaborada pela autora.

10.2.4 O que acontece com os métodos

Ao usar herança, os métodos da superclasse podem ser herdados ou sobreescritos pela subclasse, e novos métodos podem ser declarados para a subclasse. Observe que os métodos herdados não são mencionados na subclasse.

Em UML



```
public class B extends A {  
    private int at3;  
  
    public B (int b1, double b2, int b3){  
        super (b1, b2);  
        at3 = b3;  
    }  
    public void mx () {  
        //aqui o corpo do método mx  
    }  
    public void mz () {  
        //aqui o corpo do método mz  
    }  
}
```

A subclasse B tem três métodos: mx, my e mz. Observe que mx foi **sobreescrito**, my foi herdado, por isso não é **declarado** na subclasse B, e mz é um método **exclusivo** da subclasse B. Os três métodos podem ser invocados para os objetos da subclasse. Mas o método mz só pode ser invocado para os objetos da subclasse.

Figura 16 – Os métodos na super e na subclasse.

Fonte: elaborada pela autora.

10.3 Polimorfismo

É o mecanismo que nos permite chamar métodos com a mesma assinatura, mas comportamentos diferentes, usando a mesma variável objeto, e sem determinar, em tempo de compilação, a qual subclasse ou superclasse o objeto pertence. O método correto será acionado

dependendo do objeto referenciado em tempo de execução.

Para que o polimorfismo funcione devemos declarar uma variável-objeto cujo tipo é a superclasse, pois essa variável pode referenciar tanto objetos da superclasse como de qualquer subclasse. A identificação correta de qual método deverá ser invocado é realizada pela JVM, que verifica qual é o objeto real que a variável está referenciando, para acionar o método adequado.

Exemplo:

Vamos usar a seguinte hierarquia de classes:

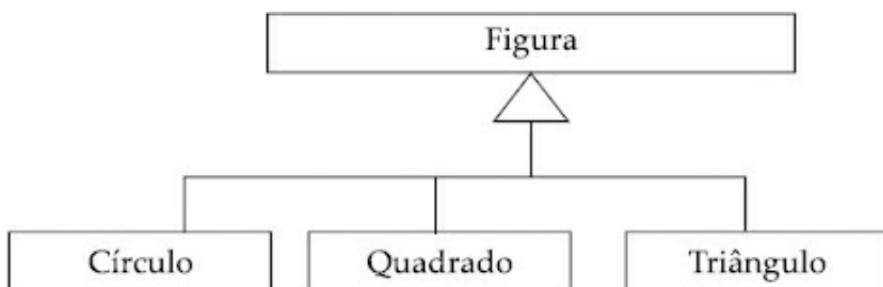


Figura 17 – Hierarquia de classes.

Fonte: elaborada pela autora.

Cada figura tem o seu método *calculaArea*, pois é sabido que o cálculo da área de um círculo é diferente do cálculo da área de um quadrado ou de um triângulo.

Suponha que temos uma classe de teste com o método *instanciaFigura*, que a partir de um sorteio instancia e retorna uma das três figuras (um círculo, um quadrado ou um triângulo).

Na classe de teste, observe como o seguinte trecho de código aproveita o polimorfismo:

```
Figura f; //declara uma variável que pode referenciar qualquer objeto  
(círculo, quadrado ou triângulo).
```

```
f = instanciaFigura(); //a variável recebe o objeto instanciado (círculo,  
quadrado ou triângulo).
```

```
double área = f.calculaArea(); // o método adequado será acionado.  
Se o objeto é um quadrado, será acionado o método que calcula a
```

área do quadrado. Se o objeto é um círculo, será acionado o método que calcula a área do círculo. Se o objeto é um triângulo, será acionado o método que calcula a área do triângulo.

Upcasting e downcasting: um exemplo

Vamos supor a superclasse **Pessoa** e a subclasse **Cliente**. A pessoa tem nome e data de nascimento, e o cliente é uma pessoa e tem, também, o valor gasto nas compras. Então, quem faz compra é o cliente, não a pessoa. Imagine uma classe de teste que instancia pessoas e clientes e armazena as referências na mesma variável (do tipo Pessoa).

Upcasting é realizado automaticamente pelo compilador: ocorre quando atribuímos um objeto da subclasse a uma variável da superclasse. Exemplo:

```
Pessoa p;  
p = new Cliente( //aqui os dados do cliente  
); //isto é upcasting
```

Downcasting não é automaticamente realizado pelo compilador: deve ser especificado pelo programador quando precisarmos chamar um método exclusivo da subclasse com uma variável da superclasse.

Exemplo: se precisarmos chamar o método *fazCompra*, que é exclusivo da classe Cliente, usando a variável p, que é do tipo Pessoa.

Assim dá erro:

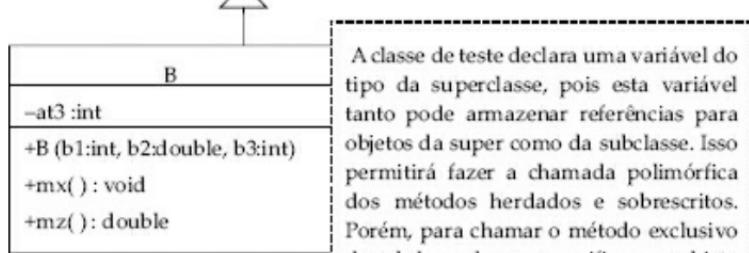
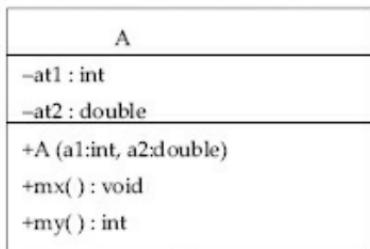
~~p.fazCompra(Teclado.leDouble("valor da compra:"));~~

Então, fazemos o downcasting, isso é, “informamos” ao compilador que p está apontando para um cliente e não para uma pessoa:

((Cliente)p).fazCompra(Teclado.leDouble("valor da compra:"));

10.4 Como instanciar objetos da super e da subclasse

A super e a subclasse, em UML



A classe de teste, em Java

A classe de teste declara uma variável do tipo da superclasse, pois esta variável tanto pode armazenar referências para objetos da super como da subclasse. Isso permitirá fazer a chamada polimórfica dos métodos herdados e sobreescritos. Porém, para chamar o método exclusivo da subclasse devemos verificar se o objeto pertence à subclasse, pois o método mz só pode ser aplicado a objetos da subclasse.

```
public class Teste {  
    public static void main (String[] args) {  
        A ob; //declara uma variável do tipo da superclasse  
        for (int i=1; i<=n; i++) {  
            if ((int) Math.random() * 10 > 4)  
                ob = new A(t.leInt(), t.ledouble());  
            else  
                ob = new B(t.leInt(), t.ledouble(), t.leInt());  
            ob.mx(); //chama mx de A ou de B, dependendo do objeto instanciado  
            ob.my(); //chama my de A independente do objeto instanciado  
            // o método mz só pode ser chamado para objetos da subclasse, pois é  
            // exclusivo da  
            // subclasse, então deve ser feita a coerção (downcasting):  
            if (ob instanceof B)  
                ((B) ob).mz();  
        } //fim do for  
    } //fim do método main  
} //fim da classe de teste
```

10.5 Classes abstratas

Uma classe abstrata é aquela que não pode ser instanciada. É uma classe que existe, unicamente, para ser herdada. Ela reúne as características comuns de todas as subclasses.

As classes utilizadas para instanciar objetos são chamadas de classes concretas. Essas classes fornecem a implementação de todos os seus métodos (ou herdam a implementação). Muitas vezes, será necessário criarmos uma classe apenas para figurar como superclasse em uma hierarquia de herança, reunindo as características comuns de todas as suas subclasses diretas ou indiretas, e sem a necessidade (pode não ser possível ou conveniente) de criar instâncias dessa classe. Essa classe deve, então, ser declarada abstrata.

Quando a classe é declarada abstrata ela pode ter um ou mais de seus métodos também abstratos, isso é, sem implementação. Isso permite adiar a implementação para a subclasse concreta.

A declaração de um método como abstrato exige a sua sobreescrita na subclasse concreta. Se uma classe estende outra que tem método abstrato e não o implementa, então ela também é considerada abstrata e deve ser declarada como tal, ou ocorre erro de compilação.

10.5.1 Sintaxe

Em Java, coloca-se a palavra-chave *abstract* antes da palavra *class*, indicando que é uma classe abstrata. Para declarar um método abstrato, coloca-se a palavra-chave *abstract* antes do tipo de retorno do método e um ponto e vírgula no final da assinatura. O método abstrato não tem corpo. Em UML, usa-se o itálico para nomes de classes e métodos abstratos. Além disso, a expressão *<abstract>* pode aparecer acima do nome da classe.

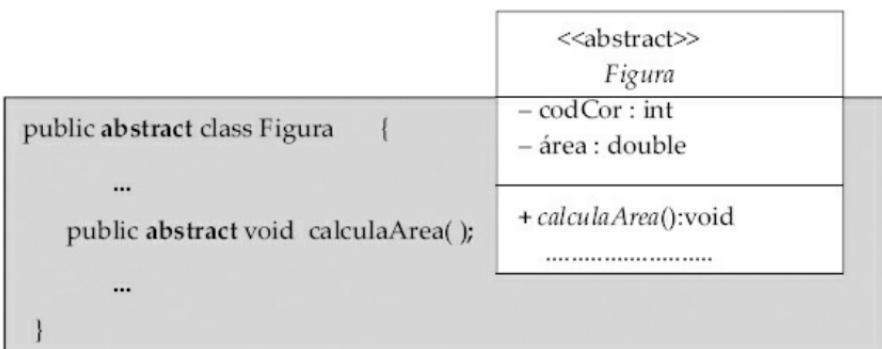


Figura 18 – Classe abstrata.

Fonte: elaborada pela autora.

CAPÍTULO 11

ARRAYS

Neste capítulo veremos a primeira estrutura de dados. Trata-se do tipo array, que oferece a facilidade de armazenar uma coleção de dados. Em Java, arrays são objetos que possuem componentes. Esses componentes devem ser todos do mesmo tipo e podem ser primitivos ou referências.

11.1 Introdução

Os tipos de dados vistos até agora foram os tipos simples, usados para representar números inteiros, números reais, caracteres etc. e até referências a objetos.

As variáveis desses tipos não oferecem facilidade para armazenarmos vários valores na memória ao mesmo tempo. Por exemplo, quando fazemos a declaração: `int num`, estamos alocando memória para apenas um número inteiro. Quando fazemos a declaração: `Circulo c`, estamos alocando memória para uma única referência a um objeto da classe Circulo.

Com arrays (também conhecidos como vetores e matrizes), poderemos ter, na memória, uma coleção de números inteiros, uma coleção de números reais e, até mesmo, uma coleção de objetos.

Em Java, os arrays são objetos cuja classe faz parte da API, e para serem utilizados devem ser declarados e instanciados. Podem ter mais de uma dimensão, mas aqui trataremos apenas de arrays unidimensionais.

11.2 Array unidimensional

O array unidimensional, também denominado vetor, é aquele que possui apenas uma dimensão.

Cada elemento da coleção é distinguido do outro por um índice (que indica a posição), pois o nome da variável é o mesmo para todos os elementos. Em Java, esse índice começa sempre em zero e deve ser sempre um número inteiro. Então, se tivermos um array a de números inteiros, de tamanho 9, teremos os seguintes elementos no array: `a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7] e a[8]`. Lê-se a índice zero, a índice um e,

assim, sucessivamente. O tamanho do array é imutável e é o conteúdo do atributo `length`, da classe `Array`, ou seja, `a.length` é o tamanho (quantidade de posições) do array `a`.

Como o array é de inteiros, cada elemento terá como conteúdo um número inteiro qualquer. Podemos armazenar, por exemplo, os números 345, 27, 1080, 0, 120, 222, 3298, -105 e 312 nos índices 0, 1, 2, 3, 4, 5, 6, 7 e 8 do array.

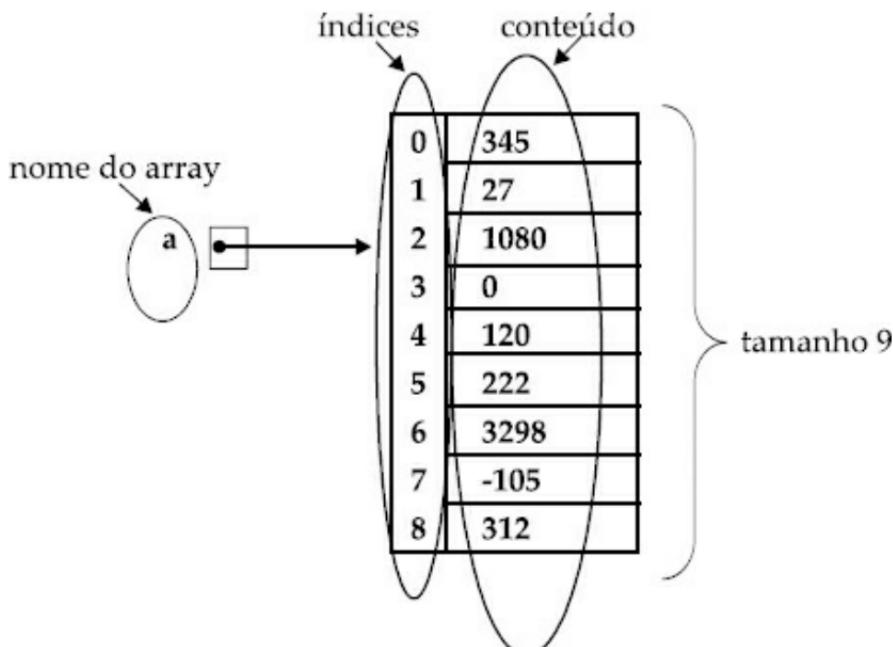


Figura 19 – Índices e conteúdo do array `a`.
Fonte: elaborada pela autora.

11.2.1 Visualização de um array unidimensional

Considere um array `a` de números inteiros, de tamanho 4, contendo os números 1529, -33, 44 e 107. Abaixo, a representação desse objeto (não esqueça que array é um objeto), na memória, usando UML:

- `a` é a variável objeto que referencia o array
- `0, 1, 2 e 3` são os índices
- `1529, -33, 44 e 107` são os números armazenados no array
- `4` é o tamanho do array
- `a[0]` armazena o número inteiro `1529`

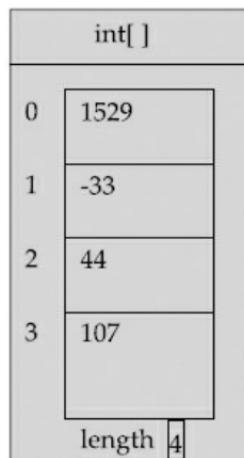


Figura 20 – Representação, em UML, do array `a`.
Fonte: elaborada pela autora.

11.2.2 Declaração

Na declaração de um array deve constar a dimensão, o tipo do componente e o nome do array. Para indicar que o array é unidimensional, usa-se um par de colchetes vazios. O tipo do componente é o identificador do tipo dos valores que serão armazenados no array. Por exemplo, para declarar um array de números inteiros, coloca-se `int`; um array de números reais, coloca-se `double` ou `float`; e assim por diante.

Finalmente, o nome do array é o identificador da variável-objeto, escolhido pelo programador, para referenciar o array.

Note que, na declaração, não especificamos o tamanho do array.

A seguir, a sintaxe da declaração.

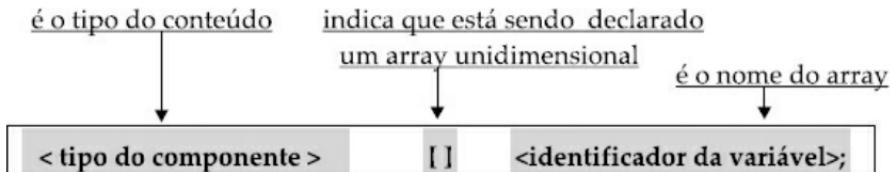


Figura 21 – Sintaxe.
Fonte: elaborada pela autora.

11.2.2.1 Exemplos de declaração

- declaração de um array unidimensional de inteiros, de nome a:
`int [] a;` ou `int a[];`
- declaração de um array unidimensional de inteiros, de nome v:
`int [] v;` ou `int v[];`
- declaração de um array unidimensional de reais, de nome notas:
`double [] notas;` ou `double notas[];`

Após as declarações acima, temos as variáveis na memória com valores null, ou seja, não estão referenciando nenhum array. A declaração apenas aloca a variável-objeto na memória e garante que a variável tem a capacidade de referenciar um objeto do tipo declarado:

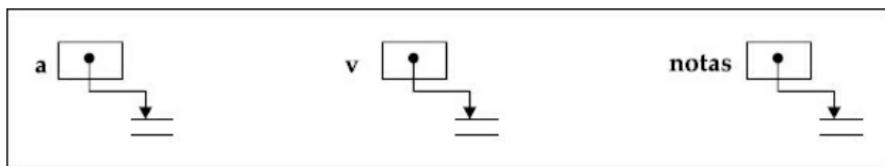


Figura 22 – As variáveis-objeto na memória.

Fonte: elaborada pela autora.

11.2.3 Instanciação

Conforme já mencionamos, os arrays são objetos e, como tais, precisam ser instanciados e atribuídos para uma variável-objeto que deve ser declarada do tipo do array para poder referenciá-lo. A instanciação do array é realizada pelo operador new, o qual aloca o array na memória com a quantidade de posições especificada e preenche todas as posições com o valor padrão, dependendo do tipo do conteúdo (se numérico, preenche com zeros; se booleano, preenche com false; se objeto, preenche com null).

Para que esse array possa ser acessado após a criação, o endereço de memória onde o array foi alocado é atribuído a uma variável-objeto do tipo array.



Figura 23 – Sintaxe da instanciação.

Fonte: elaborada pela autora.

11.2.3.1 Exemplos de instanciação

- a. instanciação de um array unidimensional de inteiros, de tamanho 5 e atribuição para a variável a:
`a = new int [5];`
- b. instanciação de um array unidimensional de inteiros, de tamanho 7 e atribuição para a variável v:
`v = new int [7];`
- c. instanciação de um array unidimensional de reais, de tamanho 4 e atribuição para a variável notas:
`notas = new double[4];`

Após as instanciações e atribuições acima, temos as variáveis na memória referenciando os arrays. Observe que agora os arrays foram instanciados (criados e alocados na memória). O conteúdo do array é zerado quando da sua criação, conforme os exemplos abaixo:

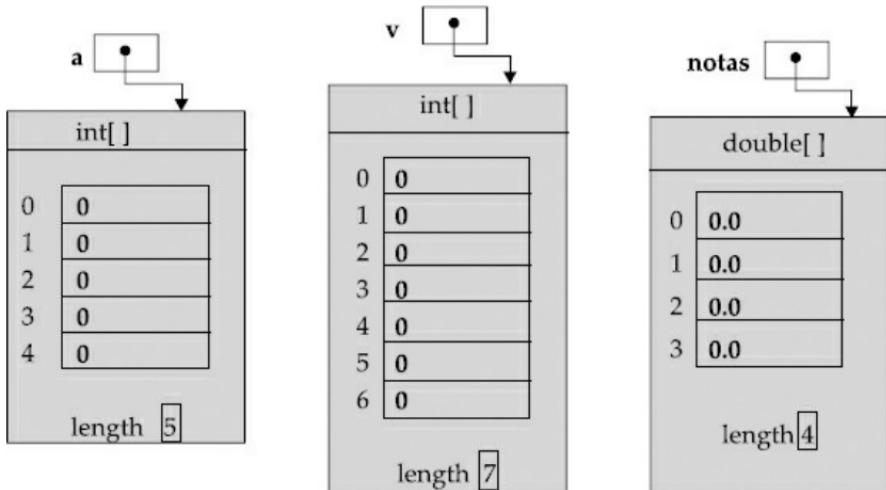


Figura 24 – As variáveis-objeto e os arrays na memória.
Fonte: elaborada pela autora.

11.2.4 Exemplos declaração e instanciação, juntas

A declaração e a instanciação de um array podem ser realizadas em uma mesma instrução, conforme os exemplos abaixo que criam os mesmos três arrays dos tópicos anteriores.

- declaração e instanciação de um array unidimensional de inteiros, de tamanho 5 e atribuição para a variável a:
`int[] a = new int [5];`
- declaração e instanciação de um array unidimensional de inteiros, de tamanho 7 e atribuição para a variável v:
`int[] v = new int [7];`
- declaração e instanciação de um array unidimensional de reais, de tamanho 4 e atribuição para a variável notas:
`double[]notas = new double[4];`

11.2.5 Como armazenar valores no array

Após a instanciação, vimos que o array é alocado na memória, porém sem conteúdo, ou seja, com valores padrões. No caso dos arrays numéricos, são criados zerados. Devemos, então, preencher o array com valores, de acordo com a sua finalidade. Se o array foi criado para armazenar notas de alunos, devemos providenciar para que essas notas sejam devidamente armazenadas. Se o array foi criado para armazenar as melhores pontuações de um jogo, então isso deve ser feito no momento propício. Enfim, os valores serão colocados no array, durante a execução do programa, quando for necessário; isso é determinado pelo programador, faz parte da lógica da aplicação e pode variar conforme o caso.

A origem dos dados também depende da aplicação. Podemos obter os dados do teclado, de um arquivo em disco, ou eles podem ser gerados pelo próprio programa. Independentemente da origem dos dados e do momento em que o array deve ser preenchido, isso dar-se-á com um comando de atribuição, atribuindo o valor para cada posição específica do array. O que devemos saber é como devemos referirmo-nos a uma posição do array. Em Java, essa referência consiste em colocar o nome do array, seguido do índice, entre colchetes. Então, para atribuir o valor 55 para a posição 3 do array a, escrevemos o seguinte comando: **a[3] = 55**.

Muitas vezes vamos querer percorrer o array inteiro, armazenando dados em todas as posições. Nesse caso, devemos usar um comando de repetição para fazer com que uma variável assuma valores inteiros (de zero até o tamanho do array menos um) e usá-la como índice.

A seguir, apresentamos vários trechos de programa que preenchem arrays, conforme o enunciado.

- **Obter via teclado (o usuário digita os valores):**

```
Scanner leia = new Scanner(System.in);
for (int i = 0; i < notas.length; i++) {
    System.out.println ("Digite uma nota:");
    notas[i] = leia.nextDouble();
}
```

- **Gerar valores aleatórios no intervalo [1,500] (usando o método random, da classe Math):**

```
for (int i = 0; i < v.length; i++)
    v[i] = 1 + (int)(Math.random() * 500);
```

- Gerar valores específicos (Ex.: múltiplos de 5 maiores que 20)

```
int mult = 20; //inicializa gerador de múltiplos de 5
for (int i = 0; i < a.length; i++) {
    mult = mult + 5; //gera múltiplo
    a[i] = mult; //armazena o múltiplo gerado no array
}
```

- Gerar valores específicos (Ex.: números pares a partir de 2)

```
int par = 0; //inicializa gerador de pares
for (int i = 0; i < a.length; i++) {
    par = par + 2; //gera par
    a[i] = par; //armazena o número par gerado no array
}
```

11.2.6 Como acessar os valores do array

Muitas vezes vamos querer percorrer o array inteiro, acessando os elementos, um a um, em todas as posições, para alterar ou não o elemento acessado. Nesse caso, devemos usar um comando de repetição para fazer com que uma variável assuma valores inteiros (de zero até o tamanho do array menos um) e usá-la como índice.

A seguir, apresentamos vários trechos de programa que percorrem arrays e acessam os elementos, conforme o enunciado.

- Exibir o array na tela:

```
System.out.println (" Array: ");
for (int i = 0; i < a.length; i++)
    System.out.print (a[i] + " ");
```

- Acumular (somar) os valores do array:

```
int soma = 0; //inicializa somador
for (int i = 0; i < a.length; i++)
    soma = soma + a[i]; //acumula
```

- Contar os valores pares armazenados no array:

```
int contPar = 0; //inicializa contador de pares
for (int i = 0; i < a.length; i++)
    if (a[i]%2 ==0)
        contPar = contPar + 1; //incrementa o contador
```

- Substituir os valores pares por 2 e os valores ímpares por 3:

```
for (int i = 0; i < a.length; i++)
    if (a[i]%2 ==0)
        a[i] = 2;
    else
        a[i] = 3;
```

11.3 Array de objetos

O array é um objeto e os seus componentes também podem ser objetos, ou seja, podemos ter um objeto que contém uma coleção de outros objetos.

11.3.1 Declaração

A declaração de um array de objetos é como a declaração de um array de tipos primitivos, o que muda é o tipo do componente. Como exemplo, temos a declaração de um array de objetos Pessoa:

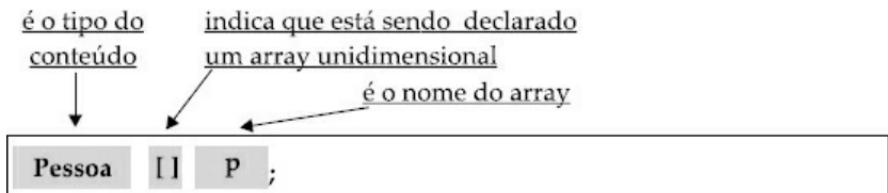


Figura 25 – Sintaxe.

Fonte: elaborada pela autora.

Após a declaração acima, temos a variável `p` na memória com valor null, isso é, não está referenciando nenhum array. A declaração apenas aloca a variável-objeto na memória e garante que a variável tem a capacidade de referenciar um objeto do tipo declarado:



Figura 26 – Sintaxe.

Fonte: elaborada pela autora.

11.3.2 Instanciação

Instanciar um array de objetos é como instanciar um array de tipos primitivos, o que muda é o tipo do componente. A instanciação do array é realizada pelo operador new, que aloca o array na memória com a quantidade de posições especificada e preenche todas as posições com o valor padrão. Nesse caso, o valor padrão é null. Para que esse array possa ser acessado após a criação, o endereço de memória onde o array foi alocado é atribuído a uma variável-objeto do tipo array. Como exemplo, a instanciação do array de objetos Pessoa, já declarado:

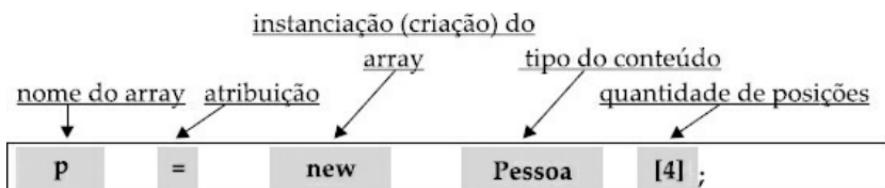


Figura 27 – Sintaxe da instanciação.

Fonte: elaborada pela autora.

Após a instanciação, temos na memória um array com capacidade de armazenar as referências de quatro objetos do tipo Pessoa. Observe que o array foi instanciado mas não foi preenchido, então, seu conteúdo é null em todas as posições. A seguir, a representação desse objeto na memória, usando UML:

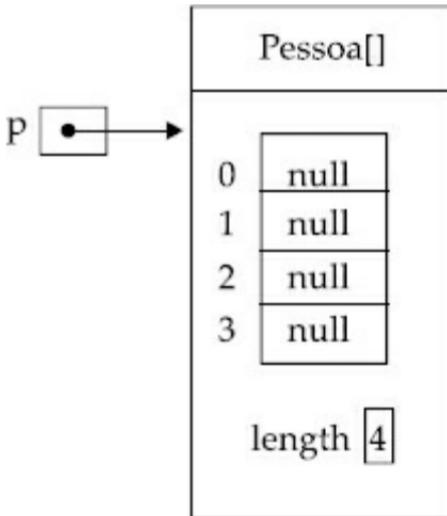


Figura 28 – Representação do array `p` em UML.
Fonte: elaborada pela autora.

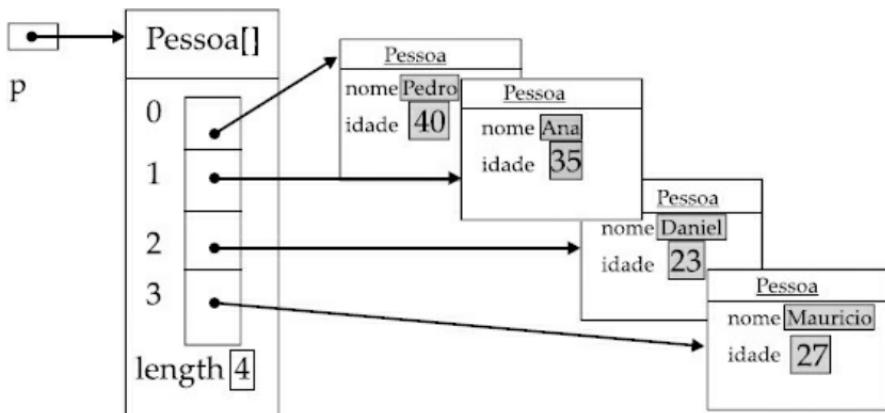
11.3.3 Como armazenar os objetos no array

Após a instanciação, vimos que o array é alocado na memória, porém sem conteúdo, ou seja, com valores `null`. Os objetos serão colocados no array, durante a execução do programa, quando for necessário; isso é determinado pelo programador, faz parte da lógica da aplicação e pode variar conforme o caso. Da mesma forma que armazenamos os valores nos arrays de tipos primitivos devemos proceder para armazenar as referências dos objetos. A título de exemplo, apresentamos alguns comandos para armazenar as pessoas no array. Como cada pessoa é um objeto, esse deve ser instanciado e apenas a sua referência é armazenada. Poderíamos usar um comando de repetição e obter os dados do teclado, mas, por clareza, faremos um a um.

```

p[0] = new Pessoa("Pedro", 40);
p[1] = new Pessoa("Ana", 35);
p[2] = new Pessoa("Daniel", 23);
p[3] = new Pessoa("Mauricio", 27);
    
```

Após a execução dos comandos acima, teremos o array p na memória, conforme a figura abaixo.



- `p` é a variável objeto que referencia o array
- 0, 1, 2 e 3 são os índices do array `p`
- `p[0], p[1], p[2]` e `p[3]` contêm as referências dos objetos `Pessoa`
- `p[2].getNome()` é “Daniel”
- `p.length` é 4
- `p[0].getIdade()` é 40

Figura 29 – Array p na memória.

Fonte: elaborada pela autora.

11.4 Array bidimensional

O array bidimensional, também denominado matriz, tem duas dimensões (as linhas e as colunas).

Cada elemento da coleção é distinguido do outro por dois índices (que indicam a linha e a coluna), pois o nome da variável é o mesmo para todos os elementos. Em Java, esses índices começam sempre em zero e devem ser sempre números inteiros. Então, se tivermos uma matriz m de números inteiros, de 3 linhas e 4 colunas, teremos os seguintes elementos na matriz: m[0][0], m[0][1], m[0][2], m[0][3], m[1][0], m[1][1], m[1][2], m[1][3], m[2][0], m[2][1], m[2][2] e m[2][3], totalizando 12 elementos. Lê-se m linha zero e coluna zero, m linha zero e coluna um, m linha zero e coluna dois e, assim, sucessivamente. As dimensões são dadas pelos atributos length, da classe Array, ou seja, m.length é a quantidade de linhas da matriz m e m[in].length é a quantidade de colunas da linha in da matriz m.

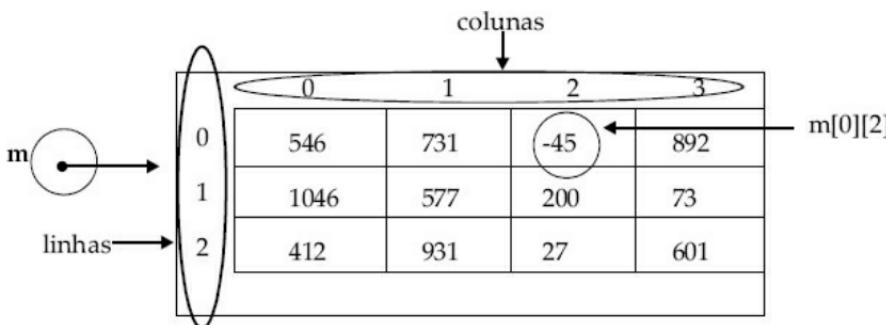


Figura 30 – Índices (colunas e linhas) e conteúdo da matriz m.

Fonte: elaborada pela autora.

11.4.1 Visualização da matriz

Considere uma matriz (array bidimensional) m de números inteiros, de 3 linhas e 4 colunas. A seguir, a representação desse objeto (não esqueça que array é um objeto) na memória, usando UML:

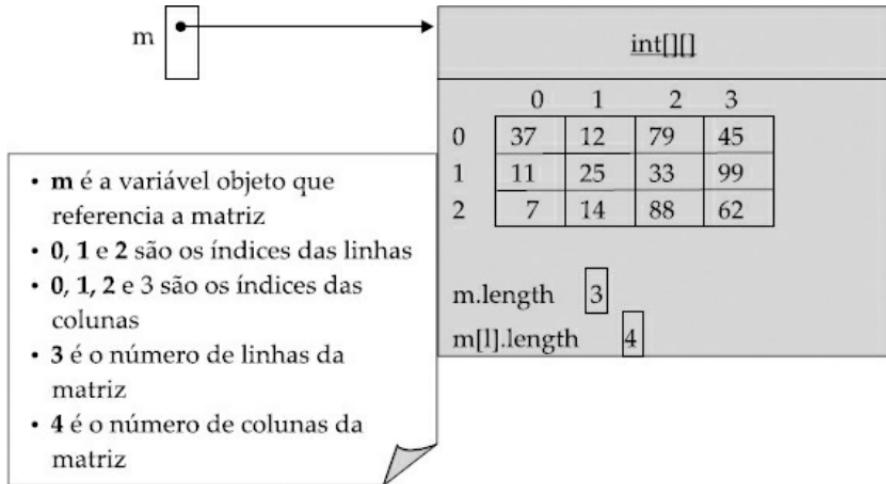


Figura 31 – Representação, em UML, da matriz.

Fonte: elaborada pela autora.

11.4.2 Declaração

Na declaração de uma matriz deve constar o tipo do componente e o nome do array. Para indicar que o array é bidimensional usam-se dois pares de colchetes vazios.

Abaixo, a sintaxe da declaração.

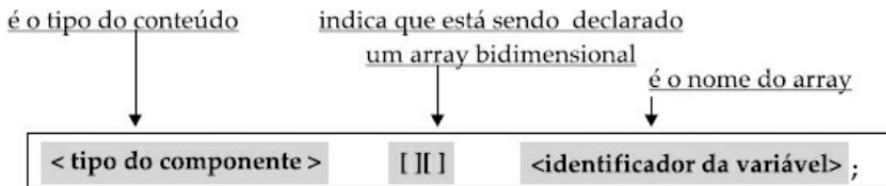


Figura 32 – Sintaxe.

Fonte: elaborada pela autora.

11.4.2.1 Exemplos de declaração

- declaração de uma matriz de inteiros, de nome m:
int [][]m; ou **int m[][];**
- declaração de uma matriz de reais, de nome mat:
double [][] mat; ou **double mat[][];**

Após as declarações acima, temos as variáveis na memória com valores null, ou seja, não estão referenciando nenhum array. A declaração apenas aloca a variável-objeto na memória e garante que a variável tem a capacidade de referenciar um objeto do tipo declarado:



Figura 33 – As variáveis-objeto na memória.

Fonte: elaborada pela autora.

11.4.3 Instanciação

A instanciação do array bidimensional é realizada pelo operador new, que aloca a matriz na memória com a quantidade de linhas e colunas especificada e preenche todas as posições com o valor padrão, dependendo do tipo do conteúdo (se numérico, preenche com zeros; se booleano, preenche com false; se objeto, preenche com null).

Para que esse array possa ser acessado após a criação, o endereço de memória onde o array foi alocado é atribuído a uma variável-objeto do tipo array.

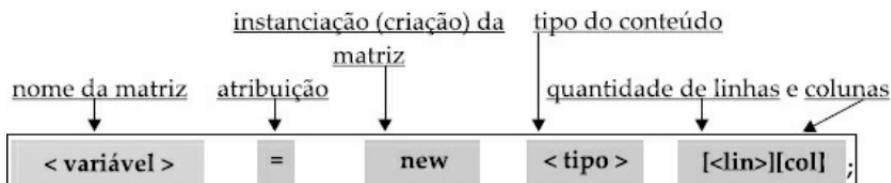


Figura 34 – Sintaxe da instanciação de matrizes.

Fonte: elaborada pela autora.

11.4.3.1 Exemplos de instanciação

- a. instanciação de uma matriz de inteiros, de 3 linhas e 4 colunas e atribuição para a variável m:
m = new int [3][4];
- b. instanciação de uma matriz de reais, de 5 linhas e 4 colunas e atribuição para a variável mat:
mat = new double[5][4];

11.4.4 Exemplos de declaração e instanciação, juntas

- a. declaração e instanciação de uma matriz de inteiros, de 3 linhas e 4 colunas e atribuição para a variável m:
int[][] m = new int [3][4];
- b. declaração e instanciação de uma matriz de reais, de 5 linhas e 4 colunas e atribuição para a variável mat:
double[][] mat = new double[5][4];

Após as instanciações e atribuições acima, temos as variáveis na memória referenciando as matrizes. Observe que agora os arrays foram instanciados (criados e alocados na memória). O conteúdo do array é zerado quando da sua criação, conforme os exemplos a seguir:

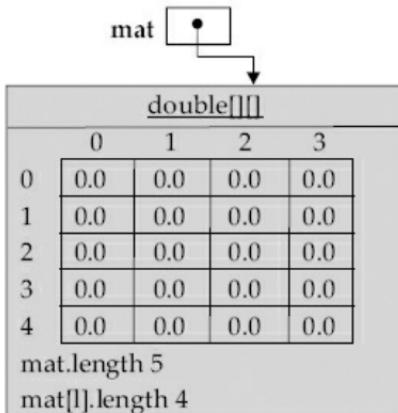
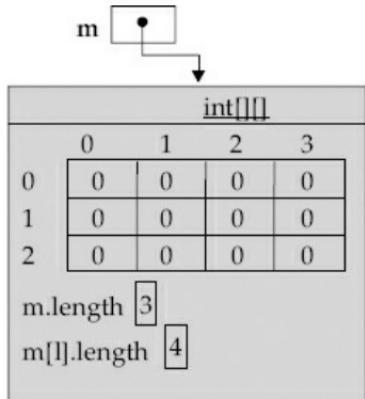


Figura 35 – As variáveis-objeto e as matrizes na memória.

Fonte: elaborada pela autora.

11.4.5 Como armazenar os valores na matriz

- **Obter via teclado (o usuário digita os valores):**

```
Scanner leia = new Scanner(System.in);
for (int i = 0; i < mat.length; i++)
    for (int j = 0; j < mat[i].length; j++) {
        System.out.println ("Digite um número: " );
        mat[i][j] = leia.nextDouble();
    }
```

- **Gerar valores aleatórios no intervalo [1,500] (usando o método Random):**

```
for (int i = 0; i < m.length; i++)
    for (int j = 0; j < m[i].length; j++)
        m[i][j] = 1+(int)(Math.random()*500);
```

11.4.6 Como acessar os valores da matriz

- **Exibir a matriz, na tela:**

```
System.out.println (" Matriz: ");
for (int i = 0; i< m.length; i++) {
    for (int j = 0; i< m[i].length; j++)
        System.out.print (m[i][j]+ " ");
    System.out.println(); //muda de linha na tela
}
```

- **Acumular (somar) os valores da matriz:**

```
int soma = 0; //inicializa somador
for (int i = 0; i< m.length; i++)
    for (int j = 0; i< m[i].length; j++)
        soma = soma + m[i][j]; //acumula
```

- **Contar os valores pares armazenados na matriz:**

```
int contPar = 0; //inicializa contador de pares
for (int i = 0; i< m.length; i++)
    for (int j = 0; i< m[i].length; j++)
        if (a[i][j] % 2 == 0)
            contPar = contPar + 1; //incrementa o contador
```

- **Substituir os valores pares por 2 e os valores ímpares por 3:**

```
for (int i = 0; i< m.length; i++)
    for (int j = 0; i< m[i].length; j++)
        if (a[i][j] %2 ==0)
            a[i][j] = 2;
        else
            a[i][j] = 3;
```

CAPÍTULO 12

EXERCÍCIOS RESOLVIDOS

Neste capítulo apresentamos vários exercícios abordando o conteúdo visto nos capítulos anteriores. Por motivos didáticos, não desenvolvemos uma aplicação real, e sim algumas classes isoladas, que poderão fazer parte de alguma aplicação, e pequenos sistemas com cenários simplificados, imitando situações da vida real.

12.1 Introdução

Desenvolver uma aplicação Java consiste em identificar, projetar e programar um conjunto de classes que se relacionam. Nos exercícios que seguem abordamos os seguintes relacionamentos entre as classes: dependência, associação e herança. O de dependência ocorre quando uma classe A usa funcionalidades da classe B; o de associação, quando uma classe A tem um ou mais atributos do tipo B. Aherança ocorre quando todo o objeto de A é um objeto de B, então A é a superclasse e B é a subclasse.

É importante, para o entendimento das classes, visualizarmos os objetos na memória em tempo de execução e, para isso, podemos utilizar diagramas de objeto, na linguagem UML, com dados fictícios adequados. Não esquecendo que cada classe tem suas responsabilidades quanto ao armazenamento de dados e aos serviços que oferece e, em tempo de execução, os objetos comunicam-se, via chamada de métodos.

Iniciamos com exercícios simples, com apenas uma classe, e depois evoluímos para exercícios que exploram os relacionamentos entre as classes.

12.2 Exercício 1 – a classe Aluno

Programar, em Java, a classe Aluno. O aluno tem nome, nota do grau a, nota do grau b e média final. A classe deve permitir que o aluno seja criado somente com o nome e zero nas notas, ou já com as duas notas: grau a e grau b. A média final deve ser calculada usando a fórmula de média ponderada com peso 1 para o grau a e peso 2 para o grau b. A classe deve oferecer as seguintes funcionalidades: acesso a todos os atributos, substituição do grau a ou do grau b.

Aluno

- nome : String
- gA: double
- gB : double
- mF : double
- + Aluno(n:String,gA:double, gB: double)
- + Aluno(n: String)
- + setNotaGa(nota: double): void
- + setNotaGb(nota: double):void
- + getNome():String
- + getMediaFinal():double
- calculaMediaFinal():void
- + substituiGrau(qualGrau:char,notaGc:double):void

```
public class Aluno {  
    private String nome;  
    private double gA;  
    private double gB;  
    private double mF;  
  
    public Aluno(String n,double gA,double gB){  
        nome = n;  
        setNotaGa (gA);  
        setNotaGb (gB);  
        calculaMediaFinal();  
    }  
    public Aluno(String n) {  
        nome = n;  
    }  
    public String getNome(){  
        return nome;  
    }  
    public double getMediaFinal(){  
        return mF;  
    }  
    public void setNotaGa (double nota){  
        if (nota>=0 && nota<=10)  
            gA = nota;  
    }  
    public void setNotaGb (double nota){  
        if (nota>=0 && nota<=10)  
            gB = nota;  
    }  
    public void substituiGrau(char qualGrau,double notaGc){  
        qualGrau = Character.toUpperCase(qualGrau);  
        if (qualGrau == 'A')  
            setNotaGa (notaGc);  
        else setNotaGb (notaGc);  
        calculaMediaFinal();  
    }  
  
    private void calculaMediaFinal(){  
        mF = gA*0.33 + gB*0.67 ;  
    }  
}
```

12.3 Exercício 2 – a classe Data

Programar, em Java, a classe Data. Os atributos, do tipo inteiro, são: dia, mês e ano. A classe deve permitir que a data seja criada com o dia, o mês e o ano recebidos via parâmetro ou a data de hoje, usando a classe GregorianCalendar, do pacote Java.util, da API. A classe deve ter as seguintes funcionalidades: oferecer acesso aos atributos e devolver a data na forma padrão dd/mm/aaaa e a data como número inteiro, na forma invertida AAAAMMDD.

```
import java.util.GregorianCalendar;
public class Data{
    private int dia;
    private int mes;
    private int ano;

    // constrói o objeto Data com os valores recebidos
    // via parâmetros
    public Data(int d, int m, int a){
        dia = d;
        mes = m;
        ano = a;
    }
    // constrói o objeto Data com a data atual
    public Data(){
        GregorianCalendar c = new GregorianCalendar();
        dia = c.get(GregorianCalendar.DAY_OF_MONTH);
        mes = c.get(GregorianCalendar.MONTH)+1;
        ano = c.get(GregorianCalendar.YEAR);
    }
    public String getDataPadrao(){
        return dia + "/" + mes + "/" + ano;
    }
    public int getDia(){
        return dia;
    }
    public int getMes(){
        return mes;
    }
    public int getAno(){
        return ano;
    }
    public int getDataInvertida(){
        return ano * 10000 + mes * 100 + dia;
    }
}
```

12.4 Exercício 3 – a empresa DiskCeva

Programar uma aplicação simplificada, em Java, para a empresa DiskCeva. A aplicação consiste em três classes: Cerveja, Cliente e Pedido.

Programar a classe Cerveja. A cerveja tem um código (que indica o tipo, de 1 a 5) e um preço. O preço mínimo de qualquer cerveja é R\$ 1,99.

```
public class Cerveja {  
    private int codigoCerveja;  
    private double preco;  
    public Cerveja(int c, double p) {  
        setCodigo(c);  
        setPreco(p);  
    }  
    public void setCodigo(int cod) {  
        this.codigoCerveja = 1;  
        if (cod>=1 && cod<=5)  
            this.codigoCerveja = cod;  
    }  
    public int getCodigo() {  
        return this.codigoCerveja;  
    }  
    public void setPreco(double pre) {  
        final double PRECO_MINIMO = 1.99;  
        if (pre>PRECO_MINIMO)  
            this.preco = pre;  
        else  
            this.preco = PRECO_MINIMO;  
    }  
}
```

Programar a classe Cliente, com os atributos privados: telefone (do tipo String), gasto acumulado (do tipo double) e quantidade de pedidos (do tipo int).

Construtor: recebe, via parâmetro, o telefone. Os outros atributos devem ser configurados com zero.

Métodos:

- *fazMaisUmPedido* – Recebe, via parâmetro, o valor gasto em um pedido e configura adequadamente os atributos.
- *zeraQuantidadeDePedidos* – Zera a quantidade de pedidos.
- *Métodos de acesso aos atributos.*

```
public class Cliente

{
    private String telefone;
    private double gastoAcumulado;
    private int numeroDePedidos;

    public Cliente(String t)
    {
        this.telefone = t;
        this.gastoAcumulado = 0;
        this.numeroDePedidos = 0;
    }

    public void fazMaisUmPedido(double valor){
        gastoAcumulado += valor;
        numeroDePedidos++;
    }

    public double getGastoAcumulado(){
        return gastoAcumulado;
    }

    public int getNumeroDePedidos(){
        return this.numeroDePedidos;
    }

    public void zeraNumeroDePedidos(){
        this.numeroDePedidos =0;
    }
}
```

Programar a classe Pedido. Um construtor que recebe o número do

pedido, o cliente, a cerveja e a quantidade de cervejas pedidas. O outro construtor que recebe o número do pedido, o telefone de um novo cliente, a cerveja e a quantidade de cervejas pedidas. Observe que qualquer um dos construtores deve chamar os métodos *calculaValorDoPedido* e *atualizaDadosCliente*.

Métodos:

- a. *calculaValorDoPedido* – Recebe o objeto Cerveja e a quantidade pedida e calcula o valor do pedido, com desconto de 10% se o gasto acumulado pelo cliente até o pedido anterior ultrapassar R\$ 100,00. O método retorna o valor calculado.
- b. *atualizaDadosCliente* – Chama o método adequado (da classe Cliente) para atualizar os dados do cliente em virtude do pedido que está sendo feito.
- c. *ganhaBrinde* – Sorteia um número de 1 a 20 e, se for sorteado um número ímpar, verifica se o cliente tem mais de 10 pedidos. Se sim, o número de pedidos do cliente deve ser zerado e o método deve retornar true, indicando que o cliente tem direito a um brinde. Caso contrário, o método retorna false.

```
public class Pedido
{
    private int numero; //número do pedido
    private Cliente cliente;
    private double valorDoPedido;

    public Pedido(int num, Cliente cli, Cerveja c, int quant){
        numero = num;
        cliente=cli;
        valorDoPedido = calculaValorDoPedido(c, quant);
        atualizaDadosCliente();
    }

    public Pedido(int num, String tel, Cerveja c, int quant){
        this (num,new Cliente(tel), c, quant);
    }

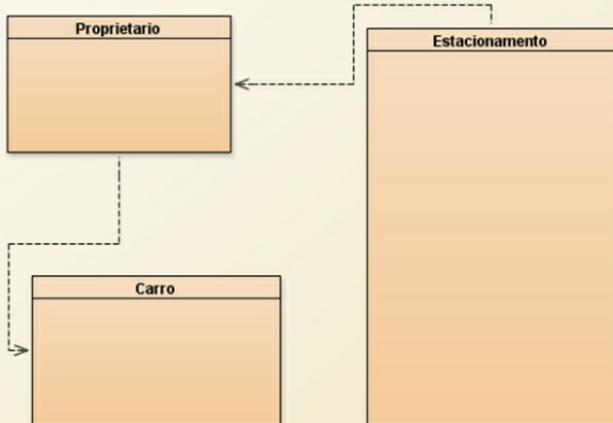
    private double calculaValorDoPedido(Cerveja c, int quant){
        double valor = c.getPreco()*quant;
        if (cliente.getGastoAcumulado()> 100.00)
            valor = valor - valor*0.1;
        return valor;
    }

    private void atualizaDadosCliente(){
        cliente.fazMaisUmPedido(valorDoPedido);

    }

    public boolean ganhaBrinde(){
        if ((1+ (int)(Math.random()*20))%2!=0)
            if(diente.getNumeroDePedidos()>10){
                cliente.zeraNumeroDePedidos();
                return true;
            }
        return false;
    }
}
```

12.5 Exercício 4 – o estacionamento do condomínio



Programar a simulação de uma pequena aplicação de um estacionamento em um condomínio. O estacionamento é constituído de vários *boxes* que são alugados aos proprietários dos apartamentos que possuem carro.

Programar a classe Carro. A classe tem como atributos privados: a placa (do tipo String) e a marca/modelo (do tipo String). Faça um construtor que receba os dois parâmetros para configurar esses atributos e os métodos get e set para cada um dos atributos.

```
public class Carro
{
    private String placa;
    private String marcaModelo;

    public Carro(String p, String mm)
    {
        placa=p;
        marcaModelo = mm;

    }

    public void setPlaca( String p )
    {
        placa = p;
    }

    public String getPlaca()
    {
        return placa;
    }
}
```

Programar a classe Proprietario. A classe tem como atributos privados: o nome do proprietário, o número do apartamento e o carro (do tipo Carro). Faça um construtor que receba os três parâmetros para configurar esses atributos e os métodos get e set para cada um dos atributos. Obs.: o método *setCarro* deve receber dois parâmetros, a placa e a marca/modelo.

```
public class Proprietario
{
    private String nome;
    private int numAp;
    private Carro car;

    public Proprietario(String n, int a, Carro c)
    {
        nome = n;
        numAp = a;
        car = c;
    }

    public void setCarro(String p, String mm1) {
        car = new Carro(p,mm);
    }

    public String getNome() {
        return nome;
    }

    public Carro getCarro() {
        return car;
    }
}
```

Programar a classe Estacionamento.

A classe tem como atributos privados:

Um array onde o índice é o número do *box*. Cada posição do array referencia um objeto do tipo Proprietário, que ocupa aquele *box*. A

numeração dos *boxes* começa em 1, então o índice 0 do array não é utilizado.

- Construtor: recebe a quantidade de *boxes* do estacionamento e cria o array com todos os *boxes* desocupados.

A classe deve oferecer as seguintes funcionalidades:

- *ocupaBox* – Recebe como parâmetro o número de um *box* e o objeto Proprietario e aloca o *box* para aquele Proprietario.
- *desocupaBox* – Recebe como parâmetro o número de um *box* e libera-o.
- *getBoxDoCarro* – Recebe como parâmetro a placa de um carro e devolve o número do *box* que o carro ocupa, ou devolve 0 se o carro não tem *box* no estacionamento.
- *temBox* – Recebe como parâmetro o nome de um proprietário e devolve true ou false, indicando se o proprietário tem algum carro em algum *box* do estacionamento ou não.
- *getQuantosLivre* – Retorna a quantidade de *boxes* livres no estacionamento.
- *isLotado* – Retorna true ou false, indicando se o estacionamento está lotado ou não.
- *getCarros* – Retorna um array contendo as placas dos carros que ocupam *box* no estacionamento, em ordem alfabética.
- *mudaDeBox* – Recebe como parâmetros o número do *box* ocupado e o número do *box* desejado pelo proprietário. Se houver disponibilidade, é efetuada a mudança. O método retorna true ou false, indicando o sucesso, ou não, da operação.
- *trocaDeBox* – Recebe como parâmetros os nomes de dois proprietários que desejam trocar de *box* entre si. O método deve providenciar a troca, se os dois proprietários possuírem *box* no estacionamento.

```

public class Estacionamento {
    private Proprietario box[];

    public Estacionamento(int max) {
        box = new Proprietario[max+1];
    }

    public void ocupaBox(int num, Proprietario p) {
        if(box[num] == null)
            box[num] = p;
    }

    public void desocupaBox(int num) {
        box[num] = null;
    }

    public Proprietario[] getTodosBox() {
        return box;
    }

    public int getQuantosLivre() {
        int q=0;
        for (int i=1; i<box.length; i++)
            if(box[i] == null)
                q++;
        return q;
    }

//continuação da classe Estacionamento

    public int getBoxDoCarro(String umaPlaca) {
        for (int i=1; i<box.length; i++)
            if(box[i] != null)
                if(box[i].getCarro().getPlacar().equalsIgnoreCase(umaPlaca))
                    return i;
        return 0;
    }

    public boolean temBox(String nP) {
        for (int i=1; i<box.length; i++)
            if(box[i] != null)
                if(box[i].getNome().equals(nP))
                    return true;
        return false;
    }

    public boolean isLotado() {
        return getQuantosLivre()== 0;
    }

    public boolean mudaDeBox(int nO, int nD) {
        if(box[nD] == null){
            box[nD] = box[nO];
            box[nO] = null;
        }
        return true;
    }

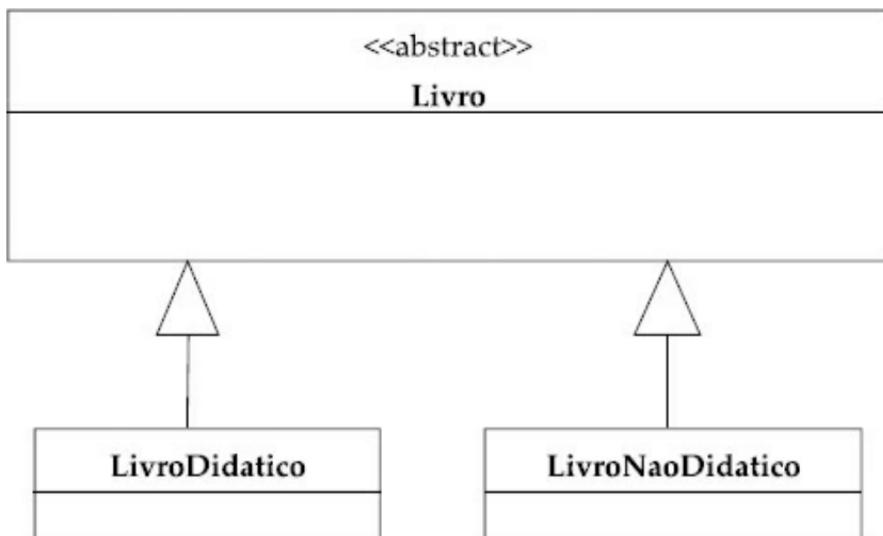
    public void trocaDeBox(String nP1, String nP2) {
        if(temBox(nP1) && temBox(nP2)){
            int boxP1=0;
            int boxP2=0;
            for (int i=1; i<box.length; i++)
                if(box[i] != null)
                    if(box[i].getNome().equalsIgnoreCase(nP1))
                        boxP1 = i;
                    else
                        if(box[i].getNome().equalsIgnoreCase(nP2))
                            boxP2 = i;
                    Proprietario salva = box[boxP1];
                    box[boxP1]=box[boxP2];
                    box[boxP2]= salva;
                }
            }

        if(box[1] != null)
            if(box[1].getNome().equalsIgnoreCase(nP1))
                boxP1 = 1;
            else
                if(box[1].getNome().equalsIgnoreCase(nP2))
                    boxP2 = 1;
                    Proprietario salva = box[boxP1];
                    box[boxP1]=box[boxP2];
                    box[boxP2]= salva;
                }
            }
        }
    }
}

```

12.6 Exercício 5 – a hierarquia de classes

Programar a hierarquia de classes:



A partir da classe abstrata Livro, programar a classe concreta LivroDidatico (uma herança da classe Livro). Para o livro didático, deve ser armazenado também o ano de publicação (do tipo int) e um atributo que permite efetuar reserva (true – reservado e false – não reservado). O empréstimo do livro didático, se reservado, só será permitido se houver mais de um volume disponível. A classe LivroDidatico deve oferecer a possibilidade de reservar e cancelar a reserva do livro e as mesmas funcionalidades da classe Livro.

Programar a classe concreta LivroNaoDidatico (uma herança da classe Livro). Para o livro não didático, deve ser armazenado também o tipo do livro (1-romance 2-autoajuda 3-outros). A classe LivroNaoDidatico deve oferecer as mesmas funcionalidades da classe Livro.

```

public abstract class Livro {
    private int codige; // código do livro
    private String titulo;
    private int q; // quantidade de volumes do livro
    private boolean disponibilidade; // array de disponibilidade
    public Livro(int c, int q, String t) {
        codige = c;
        qt = q;
        titulo = t;
        disponibilidade = new boolean[q];
        for (int i = 0; i < q; i++) {
            setDisponivelUnVolume();
        }
    }
    public int getQuantidade() {
        return qt;
    }
    public int getQrcode() {
        return codige;
    }
    public String getTitulo() {
        return titulo;
    }
    public abstract boolean faImprestimo();
    public void faLer(int i) {
        for (int j = i + 1; j < qt; j++) {
            disponibilidade[j] = true;
        }
    }
    public void setDisponivelUnVolume(int qualVolume) {
        disponibilidade[qualVolume] = true;
    }
    public void setImprestimUnVolume(int qualVolume) {
        disponibilidade[qualVolume] = false;
    }
    public boolean[] getDisponibilidade() {
        return disponibilidade;
    }
}
public class LivroDidatico extends Livro {
    private int ansPub;
    private boolean reservado;
    public LivroDidatico(int c, int q, int a, String tit) {
        super(c, q);
        ansPub = a;
        reservado = false;
    }
    public boolean faImprestimo() {
        int minimo = 1;
        if (reservado)
            minimo = 2;
        if (quantidadeVolumeDisponivel() >= minimo)
            setImprestimUnVolume(getVolumeDisponivel());
        return true;
    }
    else return false;
    }
    public void faReservar() {
        reservado = true;
    }
    public void cancelaReservar() {
        reservado = false;
    }
    public int getAnsPublicacao() {
        return ansPub;
    }
    public boolean getReservar() {
        return reservado;
    }
    public int quantosVolumesDisponivel() {
        int cont = 0;
        for (int i = 0; i < qt; i++) {
            if (disponibilidade[i])
                cont++;
        }
        return cont;
    }
    public int getVolumeDisponivel() {
        for (int i = 0; i < qt; i++) {
            if (disponibilidade[i])
                return i; // retorna o número do volume disponível
        }
    }
}
public class LivroNaodidatico extends Livro {
    private int tipos; // 1-romance 2-aut-ajuda 3-outros
    public LivroNaodidatico(int c, int q, int tString tit) {
        super(c, q, tit);
        tipos = 0;
    }
    public boolean faImprestimo() {
        if (quantosVolumesDisponivel() >= 1) {
            setImprestimUnVolume(getVolumeDisponivel());
            return true;
        }
        else return false;
    }
    public int getTipos() {
        return tipos;
    }
}

```

12.7 Exercício 6 – a classe Biblioteca

Programar a classe Biblioteca, que tem como atributos um array unidimensional de livros (didático e não didático) e a quantidade de livros armazenados (tamanho lógico). A classe deve oferecer os seguintes métodos:

- a. *pesquisaLivro* – Recebe o código de um livro e devolve o índice do array onde o livro se encontra, ou retorna -1 se o livro não está no array. Obs.: esse método deve ser utilizado (chamado) pelos outros métodos quando for conveniente.
- b. *insereLivros* – Insere vários livros no array. O método recebe, via parâmetro, a quantidade de livros que devem ser incluídos. Os dados dos livros devem ser lidos do teclado. O método devolve a quantidade de livros que foi possível incluir. O livro não será incluído se não houver mais espaço. Se o código do livro for par, é um livro didático; senão, é um livro não didático. Obs.: Cada vez que for digitado o código de um livro que já foi incluído, exibir mensagem e ler outro.
- c. *realizaDevolucao* – Recebe o código de um livro que está sendo devolvido para a biblioteca. Providencia a devolução.
- d. *realizaEmprestimo* – Recebe o código de um livro e, se o livro estiver no array, providencia para que seja emprestado, se possível. Exibe a mensagem “emprestimo realizado com sucesso” ou “falha no empréstimo”.
- e. *consultaLivro* – Recebe o código de um livro e devolve todos os dados do livro em forma de String. Devolve a string “livro inexistente” se o livro não estiver no array.
- f. *calculaPercentual* – O método recebe o tipo do livro não didático (1, 2 ou 3) e devolve o percentual de livros (sobre a quantidade de livros não didáticos), do tipo recebido como parâmetro, encontrado no array.

```
public class Biblioteca{
    private Livro[] acervo;
    private int qt;
    public Biblioteca( int quantMax){
        acervo = new Livro [quantMax];
        qt = 0;
    }
    public int pesquisaLivro (int cod){
        for (int i = 0; i < qt; i++)
            if( acervo[i].getCodigo() ==cod)
                return i;
        return -1;
    }
    public int insereLivros(int quantos){
        if (quantos > acervo.length-qt)
            quantos = acervo.length-qt;
        int cod,r;
        Teclado t = new Teclado();
        for (int i=1; i <= quantos; i++){
            do{
                cod=t.leInt("código do livro:");
                if ((r=pesquisaLivro (cod ))!= -1)
                    System.out.println("já existe");
            } while(r!=-1);
            String tit = t.leString("Título da obra:");
            if (cod%2==0)
                acervo[qt++]= new LivroDidatico(cod,t.leInt("quantos
                exemplares? "),t.leInt("ano?"),tit);
            else acervo[qt++]=
                new LivroNaoDidatico(cod,t.leInt("quantos exemplares?"),
                t.leInt("tipo?"),tit); }
            return quantos;
        }
        public void realizaDevolucao(int cod){
            int r;
            if ((r= pesquisaLivro (cod ))!= -1)
                acervo[r].fazDevolucao( );
        }
        public void realizaEmprestimo(int cod){
            int r; String msg = "Falha no empréstimo";
            if ((r = pesquisaLivro (cod ))!= -1)
                if(acervo[r].fazEmprestimo( ))
                    msg = "Empréstimo realizado com sucesso";
            System.out.println(msg);
        }
        public double calculaPercentual (int t){
            int cont = 0; int contTipo = 0;
            for (int i = 0; i < qt; i++)
                if ( acervo[i] instanceof LivroNaoDidatico){
                    cont++;
                    if ( ((LivroNaoDidatico)acervo[i]).getTipo() ==t)
                        contTipo++;
                }
            return 100*contTipo/cont;
        }
}
```


SOBRE A AUTORA

VERA MARIA DOS SANTOS ALVES

Mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS). Especialista em Sistemas de Informação pela Universidade do Vale do Rio dos Sinos (UNISINOS). Graduada no curso de Tecnólogo em Processamento de Dados pela UNISINOS.

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

Reitor: Pe. Marcelo Fernandes de Aquino, SJ

Vice-reitor: Pe. José Ivo Follmann, SJ

Diretor da Editora Unisinos: Pe. Pedro Gilberto Gomes



Editora Unisinos

Avenida Unisinos, 950, 93022-000, São Leopoldo, Rio Grande do Sul,
Brasil

editora@unisinos.br

www.edunisinos.com.br

© dos autores, 2014

2014 Direitos de publicação da versão eletrônica (em e-book) deste livro
exclusivos da Editora Unisinos.

A474p Alves, Vera.

Programação orientada a objetos [recurso eletrônico] / Vera
Maria dos Santos Alves. – 2. ed. – São Leopoldo : Ed.
UNISINOS, 2014.

1 recurso online – (EaD)

ISBN 978-85-7431-677-2

1. Programação orientada a objetos (Computadores). II.
Título. III. Série.

CDD 005.117

Dados Internacionais de Catalogação na Publicação (CIP)
(Bibliotecário: Flávio Nunes – CRB 10/1298)

Coleção EAD

Editor: Carlos Alberto Gianotti

Acompanhamento editorial: Jaqueline Fagundes Freitas

Revisão: Daniela Bittencourt

Editoração: Guilherme Hockmüller

A reprodução, ainda que parcial, por qualquer meio, das páginas que compõem este livro, para uso não individual, mesmo para fins didáticos, sem autorização escrita do editor, é ilícita e constitui uma contrafação danosa à cultura. Foi feito depósito legal.

A coleção EaD, de que faz parte este livro, é uma produção da Universidade do Vale do Rio dos Sinos para apoiar os processos de ensino e aprendizagem dos seus cursos de graduação a distância. Entretanto, o uso dessa obra não fica restrito apenas a essa modalidade de ensino, uma vez que pode servir como orientador no estudo de qualquer acadêmico. Os exemplares foram elaborados a partir da experiência de professores de reconhecido mérito acadêmico da Universidade, e traduzem a excelência dos cursos de graduação ofertados na modalidade presencial e a distância da Instituição.



MEMBRO DA
REDE DE
EDITADORES
UNIVERSITÁRIAS
DA AUSJAL
www.ausjal.org

COLEÇÃO
EAD
EDITORA UNISINOS

U
UNISINOS