



Fundação

CECIERJ

Consórcio **cederj**

Centro de Educação Superior a Distância do Estado do Rio de Janeiro

Computação II

Volume Único

Tiago Araújo Neves



**GOVERNO DO
Rio de Janeiro**

**SECRETARIA DE CIÊNCIA,
TECNOLOGIA, INOVAÇÃO E
DESENVOLVIMENTO SOCIAL**

**UNIVERSIDADE
ABERTA DO BRASIL**

**MINISTÉRIO DA
EDUCAÇÃO**



Apoio:



FAPERJ

Fundação Carlos Chagas Filho de Amparo
à Pesquisa do Estado do Rio de Janeiro

Fundação Cecierj / Consórcio Cederj

www.cederj.edu.br

Presidente

Carlos Eduardo Bielschowsky

Vice-presidente

Marilvia Dansa de Alencar

Coordenação do Curso de Engenharia de Produção

CEFET - Diego Carvalho

UFF - Cecília Toledo Hernández

Material Didático

Elaboração de Conteúdo

Tiago Araújo Neves

Diretoria de Material Didático

Cristine Costa Barreto

Coordenação de Design Instrucional

Bruno José Peixoto

Flávia Busnardo da Cunha

Paulo Vasques de Miranda

Design Instrucional

Felipe M. Castello-Branco

Biblioteca

Raquel Cristina da Silva Tiellet

Simone da Cruz Correa de Souza

Vera Vani Alves de Pinho

Diretoria de Material Impresso

Marianna Bernstein

Revisão Linguística e Tipográfica

Beatriz Fontes

Ilustração

Fernando Romeiro

Capa

Fernando Romeiro

Programação Visual

Maria Fernanda de Novaes

Produção Gráfica

Fábio Rapello Alencar

Ulisses Schnaider

Copyright © 2018 Fundação Cecierj / Consórcio Cederj

Nenhuma parte deste material poderá ser reproduzida, transmitida e/ou gravada, por qualquer meio eletrônico, mecânico, por fotocópia e outros, sem a prévia autorização, por escrito, da Fundação.

N518

Neves, Tiago Araújo.

Computação II. Volume Único / Tiago Araújo Neves. – Rio de Janeiro : Fundação Cecierj, 2018.

228p.; 19 x 26,5 cm.

ISBN: 978-85-458-0129-0

1. Computação. 2. Programação. 3. Ferramental. Título.

CDD: 001.64

Referências bibliográficas e catalogação na fonte, de acordo com as normas da ABNT.
Texto revisado segundo o novo Acordo Ortográfico da Língua Portuguesa.

Governo do Estado do Rio de Janeiro

Governador

Luiz Fernando de Souza Pezão

Secretário de Estado de Ciência, Tecnologia, Inovação e Desenvolvimento Social

Gabriell Carvalho Neves Franco dos Santos

Instituições Consorciadas

CEFET/RJ - Centro Federal de Educação Tecnológica Celso Suckow da Fonseca

Diretor-geral: Carlos Henrique Figueiredo Alves

FAETEC - Fundação de Apoio à Escola Técnica

Presidente: Alexandre Sérgio Alves Vieira

IFF - Instituto Federal de Educação, Ciência e Tecnologia Fluminense

Reitor: Jefferson Manhães de Azevedo

UENF - Universidade Estadual do Norte Fluminense Darcy Ribeiro

Reitor: Luis César Passoni

UERJ - Universidade do Estado do Rio de Janeiro

Reitor: Ruy Garcia Marques

UFF - Universidade Federal Fluminense

Reitor: Sidney Luiz de Matos Mello

UFRJ - Universidade Federal do Rio de Janeiro

Reitor: Roberto Leher

UFRRJ - Universidade Federal Rural do Rio de Janeiro

Reitor: Ricardo Luiz Louro Berbara

UNIRIO - Universidade Federal do Estado do Rio de Janeiro

Reitor: Luiz Pedro San Gil Jutuca

Sumário

Aula 1 • Conceitos básicos de programação e ferramenta.....	7
<i> Tiago Araújo Neves</i>	
Aula 2 • Variáveis e entrada/saída de dados	35
<i> Tiago Araújo Neves</i>	
Aula 3 • Desvio condicional – Parte I	65
<i> Tiago Araújo Neves</i>	
Aula 4 • Desvio condicional – Parte II	81
<i> Tiago Araújo Neves</i>	
Aula 5 • Repetição – Parte I	105
<i> Tiago Araújo Neves</i>	
Aula 6 • Repetição – Parte II	127
<i> Tiago Araújo Neves</i>	
Aula 7 • Vetores e matrizes.....	141
<i> Tiago Araújo Neves</i>	
Aula 8 • Arquivos.....	175
<i> Tiago Araújo Neves</i>	
Aula 9 • Funções – Parte I	191
<i> Tiago Araújo Neves</i>	
Aula 10 • Funções – Parte II.....	211
<i> Tiago Araújo Neves</i>	

Aula 1

Conceitos básicos de
programação e ferramental

Metas

Relembrar conceitos-chave da computação e introduzir as ferramentas utilizadas.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. instalar um sistema operacional e um compilador Java. Também aprenderá como utilizá-los para criar programas;
2. construir um programa simples para imprimir uma mensagem de texto na tela, mostrando que fazer um programa de computador é uma tarefa simples, se você souber usar as ferramentas necessárias para isso.

Introdução ao Linux

Atualmente, para operar um computador, é necessário conhecimento sobre o conjunto de programas que fornecem acesso aos dispositivos. Este conjunto de programas é chamado de Sistema Operacional (S.O.).

Existem vários sistemas operacionais no mercado, alguns pagos (Windows, iOS, etc.) e alguns gratuitos. Entre os gratuitos, um é de especial interesse para esta disciplina, o Linux na distribuição Ubuntu 14.04. Este será o sistema operacional utilizado ao longo desta disciplina.



O termo “Ubuntu” é uma antiga palavra em línguas bantu, africanas, cujo significado é “humanidade para todos” e “Eu sou o que sou devido ao que todos nós somos”. A distribuição do programa Ubuntu Linux se propõe a trazer o espírito do Ubuntu africano ao mundo do *software*: baseando-se nas premissas do *software* livre e no trabalho comunitário de desenvolvimento.

Fonte: Adaptado de UBUNTUPT, 2010.

Atividade 1

Atende ao objetivo 1

Faça uma lista dos sistemas operacionais que você conhece e/ou utiliza. Pesquise em fontes recentes qual deles é o mais utilizado. Dê a sua opinião sobre as causas de um sistema ser mais utilizado do que outros.

Resposta comentada

Existem diversos sistemas operacionais (S.O.). Certamente, você utiliza ou conhece alguns deles. Os mais conhecidos nos dias de hoje são Windows, iOS, Linux e Android, que são sistemas para uso em computadores de mesa e celulares. Atualmente, o S.O. mais utilizado é o Android. Uma das diversas razões para um sistema ser mais utilizado que outro é a existência de bons programas (aplicativos). Um S.O. é simplesmente uma base para a construção de programas. Assim, não basta ter o melhor sistema. Também são necessários bons programas. Outras possíveis razões são segurança, estabilidade, preço, etc.

Para entender melhor a função de um sistema operacional, observe a **Figura 1.1** que mostra, em forma de camadas, como se dão as interações entre os usuários, os programas e o *hardware*.

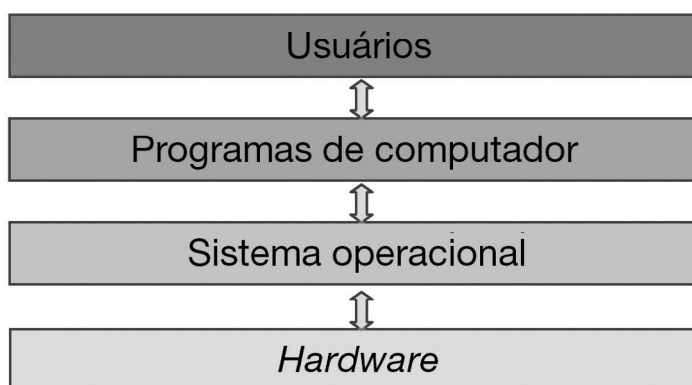


Figura 1.1: Comunicação entre usuários, programas, S.O. e o *hardware*: os usuários comunicam-se com os programas por meio de janelas ou comandos de texto, e os programas se comunicam com o sistema operacional (S.O.), que por sua vez se comunica com o *hardware*.

Como pode ser observado, o sistema operacional se comunica com o *hardware* e com os programas. Logo, pode-se perceber que uma das funções do S.O. é servir de intermediário entre os programas e o *hardware*, escondendo detalhes de *hardware* para os programas em uso.

Um bom exemplo disso é a gravação de um arquivo em disco. Ao baixar um arquivo da Internet, para salvá-lo no disco, o usuário só

precisa informar a pasta onde quer que o arquivo seja armazenado. Detalhes como o número de rotações por segundo, sentido de rotação dos discos ou posicionamento da cabeça de gravação são gerenciados pelo S.O. É importante mencionar que a **Figura 1.1** expressa o processo de comunicação usual. Em alguns casos, é possível fazer com que programas acessem o *hardware* diretamente.

Os sistemas operacionais podem ser divididos em duas camadas, chamadas *kernel* e *shell*.

A camada que é responsável por interagir diretamente com o *hardware* é chamada *kernel*, ou núcleo, do sistema. Esta camada é responsável por gerenciar as peculiaridades de cada dispositivo para que detalhes – como a quantidade de carga elétrica em um circuito, por exemplo – fiquem transparentes (invisíveis) para o usuário. É nesta camada que ficam os *drivers* de dispositivos, que são programas que fazem acesso, controle e gerenciamento de cada dispositivo, sendo estes *drivers*, em muitos casos, fornecidos pelos fabricantes de cada componente.

Para exemplificar este gerenciamento, considere um computador com dois discos (HDs). Imagine que você quer copiar dados – músicas, por exemplo – de um disco para o outro. Como usuário, tudo o que você tem que fazer é selecionar os arquivos desejados, marcar a opção de copiar, escolher a posição onde as cópias serão armazenadas e escolher a opção colar. Várias tarefas são assim executadas automaticamente pelo sistema operacional e você nem percebe. Por isso, é dito que estes detalhes são transparentes para o usuário. Neste exemplo, algumas das tarefas executadas pelo sistema operacional foram:

- mapear onde os arquivos estão armazenados no disco (posição física: mais próximo do centro ou da borda do disco);
- posicionar a cabeça de leitura na posição correta;
- transferir os dados do disco para a memória principal;
- encontrar espaço físico para armazenar os arquivos no disco de escrita;
- posicionar a cabeça de escrita na posição correta e
- transferir os dados da memória principal para o disco.

A camada *shell* é responsável por fornecer serviços e funcionalidades para o usuário e para outras aplicações.

Existem basicamente dois tipos de *shell*: os de linha de comando e os de interface gráfica. Em ambos os tipos, as funções são praticamente as mesmas. A grande diferença é o modo de apresentação:

- a camada *shell* de linha de comando usa comandos digitados através de um terminal de comandos (muito utilizado por administradores e programadores);
- a camada *shell* de interface gráfica utiliza janelas e botões (muito utilizado por usuários comuns).

Para ler um arquivo de texto do tipo PDF no computador, você usa o *shell* de interface gráfica. Ao solicitar a abertura do arquivo, um programa é aberto e o conteúdo do arquivo é exibido.

Note que, em algum ponto da janela do programa, existem botões para minimizar/fechar a janela, assim como, possivelmente, existem vários menus contendo diversas opções.

Na maioria dos casos, somente o *shell* de interface gráfica é utilizado nas operações do dia a dia. Ainda assim, você vai perceber que, para realizarmos algumas tarefas específicas, como compilar um código fonte, por exemplo, a interface de comandos é mais natural.

Frequentemente, para melhor uso de sistemas operacionais e dos computadores, é necessário o uso simultâneo dos dois tipos de *shell*, visto que cada um deles é mais adequado para a realização de determinadas tarefas. Portanto, nesta disciplina, utilizaremos os dois tipos de *shell*. Retornaremos a este ponto quando o momento for oportuno.

===== **Atividade 2** =====

Atende ao objetivo 1

A maioria dos sistemas operacionais modernos utiliza os dois *shells* de comando. Pesquise sobre as interfaces de linhas de comando dos sistemas operacionais Ubuntu e Windows. Como se chamam? São fáceis de utilizar?

Resposta comentada

Interfaces de linha de comando não são fáceis de utilizar. Diferentemente das interfaces gráficas, nas quais você clica em um determinado objeto e lhe são mostradas opções, os comandos devem ser digitados seguindo uma sintaxe específica. Caso você troque uma letra ou uma vírgula, a única resposta que terá é que o comando digitado é inválido. Por isso, no geral, somente usuários avançados utilizam este tipo de interface. As interfaces de linha de comando do Ubuntu e do Windows são chamadas, respectivamente, de terminal e prompt do MS-DOS ou simplesmente prompt.

A **Figura 1.2** representa graficamente a ideia da divisão dos S.O. em camadas. A palavra *shell*, que significa concha em inglês, expressa a ideia de algo que envolve e protege um núcleo, neste caso o *kernel* do sistema.



Em contextos diferentes, a palavra “núcleo” é expressa por meio de vocábulos diferentes da língua inglesa. Por exemplo, em geografia, tratando-se do núcleo do planeta, nós teríamos a palavra “*core*”. Assim, o “núcleo do planeta” seria referenciado como “*the planet’s core*”. Já em computação, o núcleo do sistema operacional é chamado de “*kernel*” do sistema.

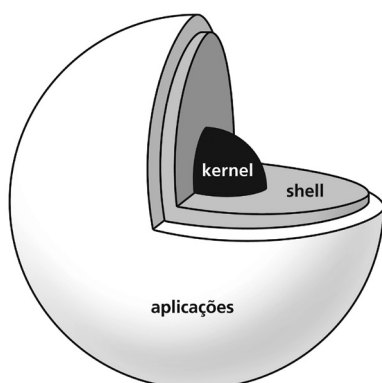


Figura 1.2: Representação gráfica das camadas do S.O: vemos o núcleo *kernel*, o *shell* e também as aplicações que são construídas sobre este. Estas aplicações são programas de alto nível, navegadores de Internet, editores de texto e organizadores de arquivos, que necessitam de serviços providos pelo *shell* para que possam executar suas tarefas.

Agora que você já tem os conceitos básicos sobre as funções de um sistema operacional, é hora de por a mão na massa e instalar o ferramental necessário.

Criando um *pen-drive* de inicialização

Para utilizar o Ubuntu em seu computador, você poderá se valer basicamente de dois modos: *dual boot* ou inicialização em um *drive* separado.

Dual boot

Muito provavelmente, este é o modo de utilização que você vai encontrar nos laboratórios utilizados nas aulas práticas ao longo desta disciplina. *Dual boot* significa a instalação de um segundo sistema operacional no seu computador. Este é, de fato, o modo mais eficiente de utilizar o sistema, visto que o sistema instalado pode utilizar os recursos de maneira otimizada.

Existem vários tutoriais na Internet sobre como fazer a instalação de um *dual boot*, e também existem vários profissionais habilitados para fazê-lo. Caso você não se sinta à vontade para realizar uma instalação *dual boot* por conta própria, procure um profissional.

Pen-drive

Nesta seção, será mostrado como criar um *pen-drive* de inicialização contendo o Ubuntu 14.04. Considere isto uma das atividades da aula, de modo que você deverá seguir os passos recomendados para criar um ambiente propício para trabalhar. Caso você utilize versões diferentes dos sistemas operacionais citados no tutorial, faça as adaptações necessárias para concluir sua tarefa. Existem diversos tutoriais sobre como fazer isso na Internet.

Algumas observações antes de começarmos a trabalhar:



Este tutorial leva em consideração uma distribuição de Linux (Ubuntu 14.04 32 bits) e uma versão de Windows (7). Novas versões dos dois sistemas surgirão ao longo do tempo, o que significa que os passos a serem feitos podem mudar ao longo dos anos.

Caso seu ambiente computacional não corresponda ao descrito nesta Aula, procure um tutorial específico para a sua versão de Windows e para a distribuição Linux de sua preferência ou procure um profissional da área.



Visando atender o maior número possível de pessoas, serão considerados sistemas com 32 bits. Sistemas 32 bits funcionam em máquinas com 64 bits, porém sistemas com 64 bits não funcionam em máquinas com 32 bits.



Como o objetivo é fornecer um mecanismo fácil de uso das ferramentas que serão utilizadas nesta disciplina, o uso otimizado dos recursos não será o foco aqui.

Sistemas 32 bits, apesar de funcionarem em máquinas com 64 bits, não conseguem utilizar todos os recursos da máquina. Se você deseja extrair o máximo de recurso da sua máquina, a melhor opção é fazer uma instalação do tipo dual boot com os sistemas correspondentes aos bits da sua máquina.

Todas as informações contidas aqui podem ser encontradas com maiores detalhes, em inglês, no site oficial do Ubuntu.



O site do Ubuntu pode ser acessado através do seguinte endereço: www.ubuntu.com

A inicialização em *drive* separado é mais simples, porém menos eficiente em termos de velocidade de resposta. Isto ocorre tipicamente porque o acesso de dados a fontes externas (hoje em dia, via USB) é mais lento do que o acesso aos discos internos (que utilizam SATA, hoje em dia).

Além disso, uma instalação em *drive* separado não pode conter configurações específicas para nenhum *hardware*, visto que este tipo de instalação é feito para rodar em qualquer máquina. Em outras palavras, como nunca se sabe em que tipo de máquina o *drive* será utilizado, ele só pode usar funcionalidades que são compatíveis com todos os dispositivos – o que pode prejudicar drasticamente o desempenho, visto que

as funcionalidades específicas de cada dispositivo é que fazem o diferencial de desempenho entre eles.

Vamos tomar placas de vídeo para exemplificar. Suponha que você tenha duas placas de desempenho similares, de fabricantes diferentes. Ambas têm em comum as tecnologias A, B e C. A diferença entre elas é a quarta tecnologia: uma usa uma tecnologia D1, e a outra, a tecnologia D2. Ao criar um *drive* de inicialização para este ambiente heterogêneo, as tecnologias diferentes não podem estar presentes, logo o *drive* só pode fazer suas tarefas utilizando as tecnologias A, B e C. Um *drive* que utiliza a tecnologia D1, por exemplo, ao ser conectado em um computador sem esta tecnologia, encontrará problemas de execução e possivelmente não concluirá com sucesso suas tarefas.

Agora, imagine fazer o mesmo (acoplar funcionalidades comuns) para todos os itens de *hardware* do seu computador. Note que, como existe uma possível perda de desempenho em cada item, a perda de desempenho global pode ser acentuada.

Contudo, a instalação em *drive* separado ainda é uma opção viável, principalmente devido a sua portabilidade. Um sistema operacional instalado deste modo pode ser executado em qualquer máquina, sem a necessidade de instalação do sistema na máquina em questão – o que requer um tempo considerável e, muitas vezes, procedimentos complexos como formatação e partição de discos.

Assim, se você quiser rodar o sistema operacional X na casa de um amigo que utiliza o sistema Y, você só precisa ter o sistema X instalado em um *drive* separado, não necessitando instalar X no computador do seu amigo. Ou seja, a máquina do seu amigo ficará exatamente como estava.

Para criar um *pen-drive* de inicialização, você vai precisar de um *pen-drive* de pelo menos 4 GB.



De posse deste *pen-drive*, a primeira coisa a fazer é o *download* da imagem do sistema operacional a ser utilizado. Para isso, procure no site do Ubuntu pela imagem (arquivo de extensão ISO) com 32 bits. Esta imagem pode ser encontrada no seguinte site:

<http://www.ubuntu.com/download/desktop>.

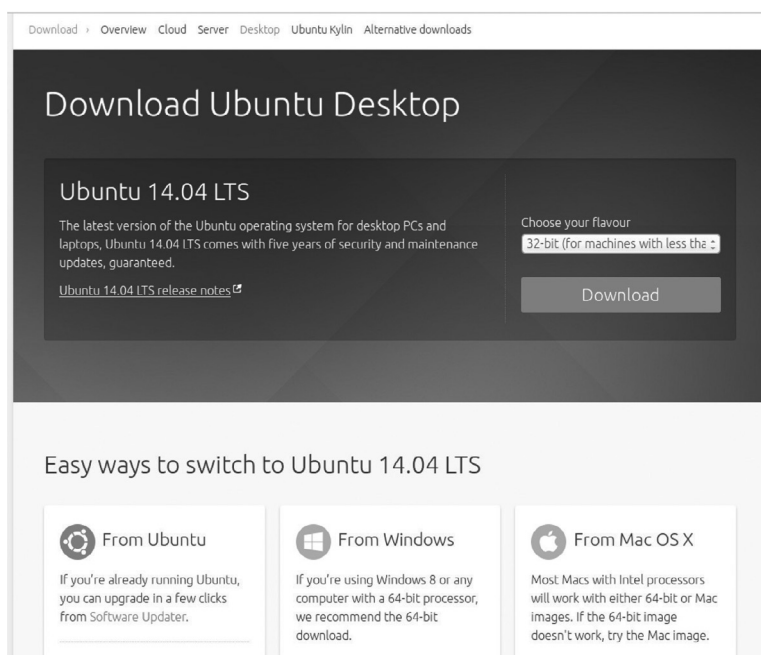


Figura 1.3: Site para download da imagem do Ubuntu: <http://www.ubuntu.com/download/desktop>.



Uma vez baixado o arquivo de imagem do sistema, você deve baixar o *software* que irá criar o *pen-drive* de inicialização.

Um dos possíveis instaladores é o “*Universal USB Installer*” (UUI), que pode ser encontrado no seguinte *link*: <http://www.pendrivelinux.com/universal-usb-installer-easy-as-1-2-3/#button>.



Neste site, basta clicar no link/imagem com a seguinte mensagem: “Download UUI”. É importante mencionar que uma versão do UUI pode estar presente na mensagem.

Escolha sempre a versão mais recente se lhe for dada a opção.

Uma vez que você tenha a imagem do S.O. e o UUI, pode começar o processo de instalação.



Coloque o *pen-drive* em uma porta USB e execute o UUI (dois cliques). Uma janela como mostrada na **Figura 1.4** dever ser exibida para você.

Esta janela mostra três campos com combo-box:

- no campo “Step 1”, selecione “Ubuntu”;
- no campo “Step 2”, clique em “Browse” e selecione o arquivo de imagem do S.O. (arquivo com extensão ISO) que você baixou;
- no campo “Step 3”, selecione a unidade em que seu *pen-drive* está habilitado. Marque a opção “Format”;
- no campo “Step 4”, arraste a barra para o máximo valor possível.

Finalmente, clique em “Create”. Quando o processo terminar, você terá o Linux instalado no *pen-drive*, ao qual chamaremos de *boot-drive* deste ponto em diante.

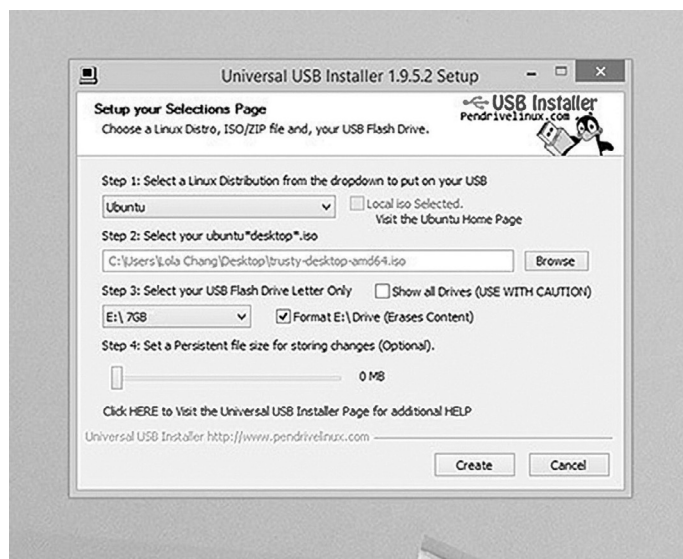


Figura 1.4: Janela principal do UUI: <http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-windows>.

Para inicializar o Linux pelo *pen-drive*, são necessárias algumas configurações no *setup* do computador. A primeira coisa a fazer é um teste de inicialização. Nos computadores modernos, muitas vezes não é necessário fazer nada, pois eles já vêm configurados de fábrica para fazer este tipo de operação. Se seu computador estiver configurado, uma imagem como a **Figura 1.5** deve aparecer.



Caso você acesse a imagem corretamente, aperte a tecla “Esc”, selecione a linguagem de sua preferência (inglês é recomendado pelo fato de o material disponível na Internet estar todo neste idioma).

Agora, selecione a opção “*Try Ubuntu without install*”, ou seja, testar o Ubuntu sem instalar. Esta opção permite rodar o S.O. a partir do *boot-drive*, sem que uma instalação no computador seja necessária. Para fazer o *boot-drive* voltar a ser um *pen-drive* convencional, basta formatá-lo (facilmente feito por qualquer S.O.).



Figura 1.5: Tela de inicialização do *boot-drive*.

Caso você não consiga visualizar a tela exposta na **Figura 1.5**, duas coisas podem ter ocorrido: (1) seu computador está configurado, porém a inicialização não é automática, ou (2) seu computador está desconfigurado. Na primeira hipótese, reinicie o computador e preste atenção nas mensagens exibidas na primeira tela que ele exibe. Uma delas deve exibir uma tecla seguida da expressão “*boot menu*” ou “*boot options*” ou alguma outra expressão relacionada com “*boot*”.

A tecla utilizada pode mudar dependendo do fabricante da placa-mãe do seu computador, contudo, F2, F8 e F10 são as mais comuns. Ao apertar a tecla certa, o *pen-drive* deve ser exibido como opção. É possível que o *pen-drive* esteja em mais de uma opção. Neste caso, a escolha certa é a opção que contiver a expressão “*Legacy*” ou a que NÃO contém a sigla “U.E.F.I.”.

Caso o *pen-drive* não seja exibido em nenhuma das opções, significa que seu computador está desconfigurado.



Acesse o *setup* de inicialização da sua máquina e habilite a inicialização por USB.

Em alguns casos, é necessário estabelecer uma prioridade na inicialização. Neste caso, coloque o *boot* por USB como

primeira tentativa. O *setup* pode ser acessado mediante observação das mensagens na primeira tela exibida durante a inicialização. Uma destas mensagens deve exibir uma tecla e a opção “*Setup*”.

Várias são as possíveis teclas de acesso ao *setup*, porém a tecla “*Del*” é a mais comum para esta operação. Cada fabricante utiliza um sistema de *Setup* diferente. Se você não sabe como fazer esta configuração, procure um profissional.

Uma vez que você inicializou o Ubuntu pelo *boot-drive*, deve ser exibida para você uma tela parecida com a **Figura 1.6**. Esta é a área de trabalho do Ubuntu. Nela você pode ver diversos botões e dois ícones. Um dos ícones é a opção de instalação em *dual boot* do Ubuntu. O outro é só um atalho para o gerenciador de arquivos.

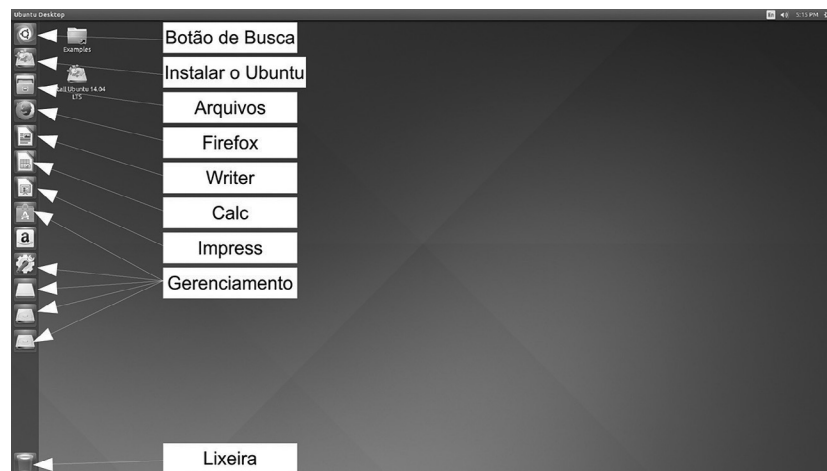


Figura 1.6: Tela inicial do Ubuntu. Os botões, de cima para baixo, são os de Busca (onde podem ser encontrados todos os programas); Instalação (para instalar o Ubuntu); Arquivos (para acessar e gerenciar os arquivos); Firefox (para navegar na Internet); Writer (para editar textos); Calc (para criar planilhas eletrônicas); Impress (para criar apresentações de *slides*). Os demais botões são opções para gerenciamento do sistema.

Para finalizar a configuração do *boot-drive* é necessário instalar o compilador que iremos utilizar ao longo da disciplina. A definição de **compilador** e os detalhes de seu funcionamento serão mostrados mais adiante nesta aula. Por enquanto, vamos nos preocupar apenas com a sua instalação.



Primeiro, teste sua conexão com a Internet acessando um *site* qualquer. Caso não tenha sucesso, procure configurar sua rede.

Caso sua conexão com a Internet esteja funcionando, clique no botão de busca e digite “terminal”. Clique na opção que exibe “Terminal” como nome. A **Figura 1.7** exibe este passo. Nela são exibidas duas telas mostrando o processo de busca pelo terminal e a janela do terminal aberta.



Figura 1.7: Acessando o terminal via o botão de busca. O terminal é o *shell* de comando do Ubuntu. É nele que será executada a maior parte dos comandos a serem executados.



Na janela do terminal, digite o seguinte comando: “`sudo apt-get update`” – tudo em letras minúsculas – e espere a execução total do comando, que pode levar bastante tempo.

Este comando serve para atualizar o catálogo de *softwares* disponíveis.



Em seguida, digite “`sudo apt-get install openjdk-7-jdk`”, também em letras minúsculas. Este comando serve para instalar o “*Java Development Kit*” (JDK) versão 7. Obs: pode haver mais de uma versão ou a versão 7 pode não ser a mais atualizada disponível para o seu sistema. Neste caso, procure na Internet a versão mais atual do JDK disponível para instalação no seu sistema operacional.

Neste caso, procure na Internet a versão mais atual do JDK disponível para instalação no seu sistema operacional.



Quando a execução do comando terminar, verifique se o processo foi bem-sucedido. Para isso, digite “`javac -version`” no terminal. Deverá aparecer a versão do JDK instalado. Por exemplo, “`javac 1.7.0_65`”.

Caso você não consiga realizar a instalação do JDK, procure um profissional.

Com todos os passos executados com sucesso, você tem em seu *boot-drive* tudo o que é necessário para fazer os exercícios desta disciplina. Sempre que quiser acessar o Linux, basta iniciar o computador com o *boot-drive* conectado, pedindo que a inicialização seja feita por ele.

Uma vez que o sistema esteja totalmente carregado, você poderá trabalhar normalmente, como se o sistema estivesse instalado em seu computador.



É importante mencionar que o *boot-drive* não pode ser retirado enquanto o sistema operacional estiver ativo. Para retirá-lo, desligue o computador e, só depois, retire-o da porta USB.

Agora que todos os programas necessários estão instalados, você já pode começar a criar os seus programas. Porém, algumas observações precisam ser feitas antes de começarmos.

IDE

Acrônimo de *Integrated Development Environment*, que significa Ambiente de Desenvolvimento Integrado. IDEs são programas que auxiliam o programador no desenvolvimento e na manutenção de *softwares*. Para a linguagem Java, dois dos mais conhecidos são Eclipse e Netbeans, que estão presentes tanto em Linux como em Windows.

Para a criação de códigos-fonte você pode utilizar qualquer **IDE** de sua preferência (Netbeans, Eclipse, etc.) e qualquer distribuição do JDK (Oracle, open, etc.). Porém, por questão de padronização, utilizaremos para esta disciplina um editor de texto padrão (Gedit) e um JDK padrão (openJDK), que podem ser encontrados em qualquer distribuição Linux. Também é importante mencionar que estas serão as ferramentas disponíveis para você em laboratório para aulas práticas e provas. É possível que outras ferramentas estejam disponíveis, mas não é garantido. Portanto, você deve aprender a utilizar as ferramentas padrão.

Caso você prefira utilizar o Windows em sua casa, você encontrará compiladores Java com facilidade. Qualquer editor de texto (Word, Notepad, etc.) padrão pode ser utilizado para editar os códigos ou você pode usar IDEs para Windows. Entretanto, é preciso notar que não há garantias de que você encontrará em laboratório (para aulas presenciais e provas) exatamente as mesmas ferramentas que você estiver utilizando, caso você opte pelo Windows.

Como fazer computação

A computação, que nada mais é do que utilizar um computador para resolver um problema, é sempre feita através de programas de computador. Isto significa que, para resolver um problema computacionalmente, alguém deve construir um programa para resolvê-lo. Um exemplo é o uso do computador para escrever textos. Hoje em dia, há várias opções de editores, mas alguém (empresa ou programador) teve que construí-los para que nós pudéssemos utilizá-los.

Um programa de computador, por sua vez, nada mais é do que um **algoritmo** escrito de uma forma que o computador consiga interpretar.

Como você já deve saber, o computador só entende um conjunto de instruções compostas por zeros e uns (instruções binárias). Contudo, construir um programa usando linguagem binária é uma tarefa árdua para seres humanos. Por isso, hoje em dia, utilizamos as chamadas linguagens de programação que são linguagens mais naturais para seres humanos e que podem ser transformadas para a linguagem de máquina.

Para relembrar, a solução de um problema começa com uma ideia para sua resolução, e esta ideia deve ser expressa através de um algoritmo. Você teve contato com estes dois primeiros passos do processo na disciplina Computação I. Uma vez construído o algoritmo, deve-se construir o código-fonte, que nada mais é do que um algoritmo escrito seguindo a sintaxe de uma linguagem de programação.

O código-fonte é, então, submetido a um **compilador**, que é um programa que “traduz” o código-fonte para linguagem de máquina. O resultado final da “tradução” é um programa que pode ser executado nos computadores. Nesta disciplina, você terá contato com todos os passos do processo de produção, ou seja, serão apresentados problemas para os quais você deverá bolar um algoritmo, transcrevê-lo para uma linguagem de programação, compilar o código-fonte e executar o programa resultante.

Algoritmo

Conjunto finito de passos bem definidos para resolver um problema.

Compilador

É um programa capaz de traduzir algoritmos escritos em uma determinada linguagem de programação de alto nível (Java, C ou Fortran, por exemplo) para uma linguagem de baixo nível. A maioria dos compiladores modernos consegue gerar diretamente os programas binários (linguagem de máquina). Alguns compiladores antigos conseguiam somente gerar linguagens intermediárias, que posteriormente eram convertidas para linguagem de máquina com a ajuda de outros programas (programas montadores).

A **Figura 1.8**, a seguir, mostra a cadeia de produção de um programa.

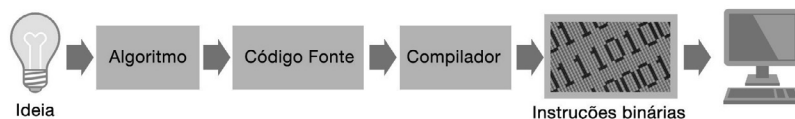


Figura 1.8: Processo de criação de *software*. O processo começa com uma ideia para resolver o problema em questão. Formaliza-se esta ideia através da criação de um algoritmo. Em seguida, escolhe-se uma linguagem de programação e traduzimos o algoritmo para esta linguagem, criando o código-fonte. Submetemos o código-fonte para um compilador, que irá transformá-lo em instruções binárias que, por fim, poderão ser executadas em um computador.

Nesta disciplina, utilizaremos a linguagem de programação Java para construir nossos programas. Java é uma linguagem compilada e interpretada, o que significa que o final do processo de compilação não é um programa, mas sim um arquivo contendo instruções em uma linguagem intermediária, chamada *bytecode*. Um programa, chamado de interpretador, é utilizado para ler este arquivo de instruções e, em seguida, executar os comandos.

Tudo o que você precisa ter no computador que irá executar o código é um interpretador bem configurado. Se você conseguiu instalar o JDK como propusemos na seção anterior, você também instalou, automaticamente, um interpretador. Logo, rodar os programas que você criar não será um problema.

Atividade 3

Atende ao objetivo 1

As linguagens de programação podem ser compiladas ou interpretadas, ou as duas coisas ao mesmo tempo, como é o caso de Java. No mundo moderno, qual seria a principal vantagem de uma linguagem interpretada sobre uma linguagem compilada na execução de um projeto? E a principal vantagem de uma linguagem compilada sobre uma interpretada?

Resposta comentada

A grande vantagem de se utilizar uma linguagem interpretada hoje em dia é que os programas feitos deste modo podem ser executados em qualquer tipo de computador/S.O. (Windows, Linux, Android, etc.). Isto faz com que os programas tenham uma portabilidade maior, bem como um alcance maior, visto que não são feitos para um sistema específico. A grande vantagem de utilizar uma linguagem compilada é o desempenho. Como o resultado final é um programa com instruções binárias para ambiente específico (computador/S.O.), é possível tirar o máximo de proveito das especificidades deste ambiente.

Fazendo o seu primeiro programa

Agora que os principais conceitos foram revistos e o ambiente (computador e *softwares*) está configurado, está na hora de fazer seu primeiro programa.



Para isso, clique no Explorador de Arquivos e crie uma pasta para colocar seus exemplos.

A pasta inicial do Explorador é chamada de pasta padrão ou pasta *home*. Para criar uma pasta, você só precisa procurar a opção na interface (File > New Folder ou Arquivo > Nova Pasta) ou utilizar o botão direito do mouse. A **Figura 1.9**, a seguir, mostra estes dois passos.

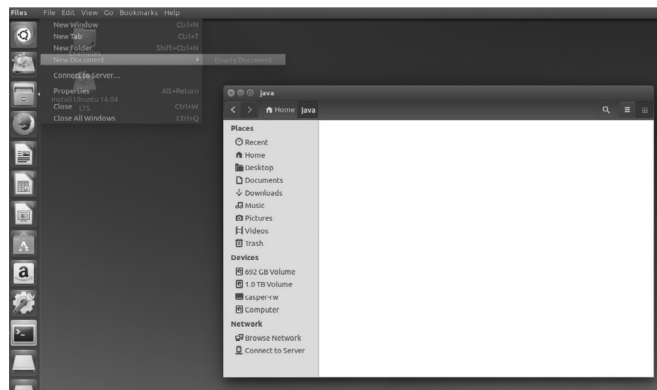


Figura 1.9: Criação de pasta. Crie uma pasta com nome simples (sem acentos ou espaços), pois você precisará utilizar este nome com certa frequência. Uma sugestão para o nome da pasta é “java”.



Em seguida, acesse a pasta (dois-cliques) e crie um “Documento Vazio” com o nome “PrimeiroPrograma.java”. Você pode fazer isso através da interface do explorador (File > New Document > Empty Document ou Arquivo > Novo Documento > Documento Vazio) ou com o botão direito do mouse. Este arquivo conterá o código-fonte do seu primeiro programa. A **Figura 1.10**, a seguir, mostra este passo.

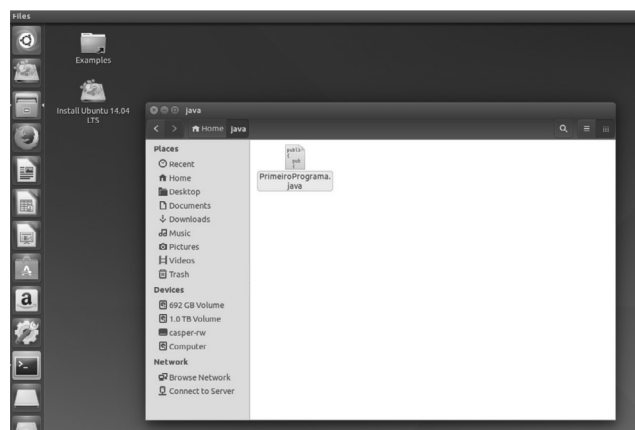


Figura 1.10: Criação de arquivo. O nome do arquivo é importante, pois deve coincidir com o nome da classe. Não use acentos, nem espaços. Comece as palavras do nome com letras maiúsculas e sempre termine com “.java”.



Abra o arquivo (dois-cliques) e digite o texto que está na **Figura 1.11**. O texto deve estar exatamente igual ou você não conseguirá dar prosseguimento ao exercício.

```

PrimeiroPrograma.java x
1 public class PrimeiroPrograma
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("O primeiro programa foi executado com sucesso!");
6     }
7 }
8

```

Figura 1.11: Primeiro código-fonte. O objetivo deste programa é imprimir uma pequena mensagem na tela para certificar que tudo está funcionando corretamente.



Java é sensível às letras maiúsculas. Portanto, o uso de letras maiúsculas no lugar errado pode levar a um programa que não compile. Use letras maiúsculas e minúsculas nos lugares certos.

Quando você terminar de digitar, confira para ver se você não digitou algo errado e salve o arquivo. Seu primeiro código-fonte está criado. Agora você deve compilá-lo e executá-lo.



Para compilar o seu código, você deve abrir uma janela do terminal. Ao abrir a janela, o terminal aponta para a pasta *home*. Logo, você deve “posicionar” o terminal na pasta onde está o seu arquivo. Para fazer isso, você deve digitar o seguinte comando “`cd nomeDaPasta`”.

O comando `cd` é o comando de mudança de diretório. Ele faz o terminal apontar para a pasta indicada pelo nome digitado junto com o comando. Se você seguiu as recomendações anteriores, sua pasta se chama “`java`”. Logo, você deve digitar “`cd java`”.

Uma vez que o terminal esteja apontando para o diretório correto, você pode compilar e executar o programa. Para compilar, digite “javac PrimeiroPrograma.java”. Se alguma coisa der errado, será exibida uma mensagem de erro no terminal. Se nenhuma mensagem for exibida, quer dizer que o processo de compilação foi concluído com sucesso.

Se você olhar a pasta através do explorador, verá que um novo arquivo – chamado “PrimeiroPrograma.class” – foi criado. Este é o arquivo que contém as instruções a serem interpretadas. A **Figura 1.12**, a seguir, exhibe estes passos.

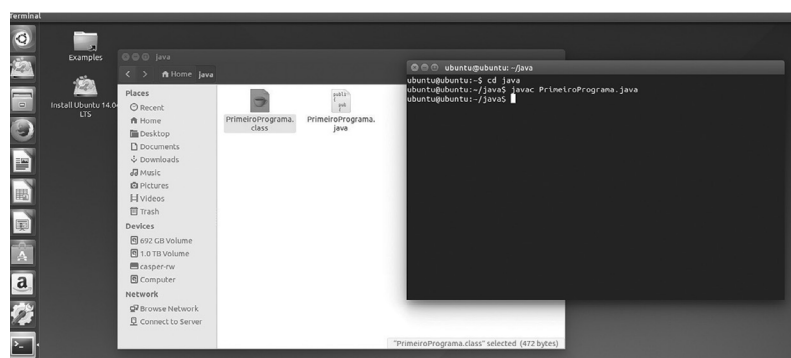


Figura 1.12: Compilando o primeiro programa. Caso algo dê errado, as mensagens de erro serão exibidas na tela. Caso nenhuma mensagem seja exibida, a compilação foi concluída com sucesso.

Para executar o programa, você deve utilizar, no terminal, o comando “java PrimeiroPrograma”.



Figura 1.13: Execução do primeiro código-fonte. Para executá-lo, você deve utilizar, no terminal, o comando “java PrimeiroPrograma”. A mensagem “O primeiro programa foi executado com sucesso” será exibida.

É importante mencionar que qualquer erro, mesmo que mínimo, leva a outros erros durante o processo de compilação. A **Figura 1.14** mostra o erro exibido quando a letra “s” da palavra *System* é escrita com letra minúscula.

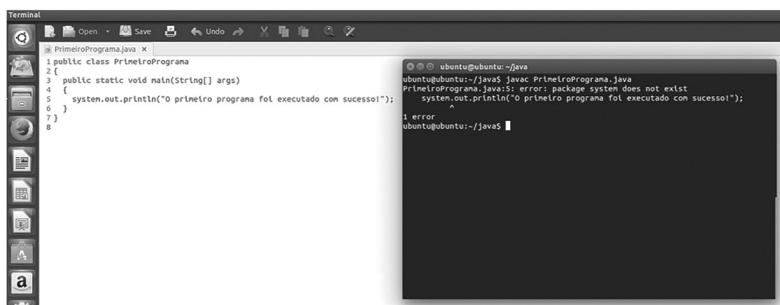


Figura 1.14: Erro de digitação. Aqui vemos o erro exibido quando a letra “s” da palavra *System* é escrita com letra minúscula.

Se tudo correu bem, você conseguiu compilar e executar seu primeiro programa.



Para finalizar esta aula, o programa exibido será descrito parte a parte. Não se preocupe se você não entender tudo imediatamente. Muitos conceitos ficarão mais claros à medida que você for entendendo melhor a linguagem Java.

- As chaves (“{” “}”) são os delimitadores de Java. Elas marcam tanto o início (“{”) quanto o fim (“}”) de várias estruturas, dentre as quais estão as estruturas de classes e funções, mostradas na **Figura 1.11**.
- No início do código, temos as palavras reservadas “public” e “class” seguidas de um identificador “PrimeiroPrograma”. Esta linha está definindo uma classe, chamada PrimeiroPrograma como pública. Praticamente tudo em Java é feito através da definição de classes, logo, mesmo para um programa simples como o que fizemos, é necessário criar uma classe. O conceito de classe, seu uso e seus benefícios serão abordados em momento oportuno. Por enquanto, trate a criação da classe como sintaxe obrigatória.
 - O nome da classe precisa ser idêntico ao nome do arquivo em que ela se encontra para as classes públicas, por isso o nome da classe e do arquivo são os mesmos.
 - Uma classe pública é uma classe que pode ser acessada por outras classes. Você verá a utilidade de classes públicas quando estiver

estudando modularização, mais adiante, nesta disciplina.

- Na linha 3, há a expressão “public static void main(String[] args)”. Esta linha trata da declaração de uma função especial de todo programa Java, chamada *main*. Um programa Java pode ser composto de diversas funções e classes, contudo, pelo menos uma classe e uma função são obrigatórias. A classe tem nome livre, mas a função obrigatória tem nome fixo *main*.

Quando se executa um programa java, o interpretador busca a função *main* como ponto inicial. Ou seja, esta é sempre a primeira função a ser executada.

- As palavras “*public*” e “*static*” são modificadores que dizem que esta função pode ser vista por outras classes e que ela não depende da criação de objetos (desta classe). Estas duas questões ficarão mais claras quando você estiver estudando programas com múltiplas classes. Por enquanto, trate-as como sintaxe obrigatória.
- O termo *void* indica o tipo desta função. Como você deve se lembrar das aulas de Computação I, o tipo de uma função é o tipo de dado que a função retorna. Quando não queremos que a função tenha retorno, dizemos que o tipo desta função é *void*, ou seja, esta função não retorna nada. Esta parte ficará mais clara quando você estiver estudando funções.
- Após o nome da função, está a lista de argumentos que ela recebe entre parênteses “(String[] args)”.
 - A função *main* sempre recebe como argumento um vetor de textos que, por convenção, é chamado de *args*, mas poderia receber qualquer nome. O termo *args* é um diminutivo para *arguments*, que significa argumentos.
 - O termo *String* se refere a um tipo de java que serve para armazenamento de textos, os colchetes “[]” indicam que o parâmetro é um vetor, e “args” é o nome do parâmetro (dentro do contexto de funções, parâmetros e argumentos são sinônimos).

Estas sintaxes ficarão mais claras quando você estiver estudando vetores, por enquanto, trate-as também como sintaxe obrigatória.

- Logo após a definição da função, existe, na linha 4, um “{” que indica o início da função *main*. Esta função contém um único comando que

é a chamada da função `System.out.println` (“*O primeiro programa foi executado com sucesso*”). A função `System.out.println()` é uma das várias funções de impressão em Java. Esta função imprime o texto que está descrito entre os parênteses. Neste caso, a expressão de texto: “*O primeiro programa foi executado com sucesso*”. Toda expressão de texto em Java deve ser escrita entre aspas duplas.



Uma aspa dupla (") não é composta por duas aspas simples (').

Deste modo, use uma aspa dupla no início de uma expressão de texto e outra aspa dupla no final da expressão. Substituir uma aspa dupla por duas aspas simples é um erro de sintaxe computacional muito comum, especialmente devido ao fato de que estes dois caracteres costumam partilhar a mesma tecla do seu teclado.

Conclusão

Ao longo desta aula, muitos conceitos foram vistos, e outros tantos foram recordados. Você aprendeu como instalar um sistema operacional em um *pen-drive*, a instalar programas utilizando terminal de linhas de comando e até compilou e executou seu primeiro programa, o que já é bastante para uma primeira aula. Toda a explicação sobre a sintaxe Java pode não ter ficado totalmente clara neste momento, devido ao fato de muitos dos conceitos serem novos (classe, modificadores públicos) e outros, complexos para iniciantes (funções, argumentos, tipo *void*). Contudo, você verá que muitos destes conceitos já foram estudados em seus fundamentos, em aulas anteriores, e as coisas ficarão mais claras à medida que você estudar e praticar.

Ao longo deste curso, você aprenderá a lidar com todas as peculiaridades da sintaxe computacional progressivamente; de modo incremental. Isto significa que, às vezes, você usará uma ferramenta sem entendê-la plenamente, mas, ao longo do curso, esta ferramenta será detalhada; progressivamente, de modo que seu entendimento sobre ela seja construído de maneira cumulativa.

Assim, todos os detalhes serão revistos mais adiante, quando o momento for oportuno. De qualquer forma, sempre é válido fazer uma releitura do texto para afiar o conhecimento, preparando-se para os próximos passos.

===== **Atividade final** =====

Atende aos objetivos 1 e 2

Tente refazer, agora como atividade, as partes de criação, compilação e execução do código-fonte apresentado na **Figura 1.11**. Só que agora o seu arquivo deve ter um nome diferente (use “teste.java”, por exemplo). Mantenha o código-fonte exatamente como está e tente compilá-lo.

Você deve obter uma mensagem de erro. Procure na Internet o significado desta mensagem e como resolver o problema para que o código do arquivo “teste.java” passe a funcionar exatamente como o do “PrimeiroPrograma.java”.

Resposta comentada

Ao criar um arquivo chamado “teste.java”, colocar nele o texto da **Figura 1.11** e compilá-lo, obtemos uma resposta similar a esta:

*teste.java:1: error: class PrimeiroPrograma is public,
should be declared in a file named PrimeiroPrograma.java*

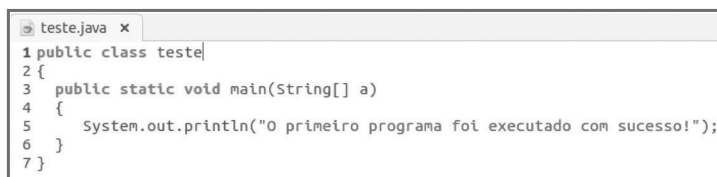
public class PrimeiroPrograma

^

1 error

Ao procurar na Internet, você receberá a informação de que toda classe pública deve ser armazenada em um arquivo que tenha o mesmo nome da classe. Logo, se você quer um arquivo que se chame “teste.java”, a

classe pública dentro deste arquivo também deve se chamar “teste”. Desse modo, para que o arquivo “teste.java” possa ser compilado e executado, o nome da classe deste arquivo deve ser alterado para teste. O código final do arquivo “teste.java” deve ser exatamente como descrito abaixo:



```
1 public class teste
2 {
3     public static void main(String[] a)
4     {
5         System.out.println("O primeiro programa foi executado com sucesso!");
6     }
7 }
```

Resumo

Nesta aula, nós fizemos uma breve revisão sobre sistemas operacionais. Você também aprendeu a criar um *boot-drive*, usando o Ubuntu como sistema operacional e a instalar o compilador que será necessário para a disciplina. Foram lembrados os passos para resolução de problemas utilizando o computador e um primeiro programa, para imprimir mensagens na tela, foi construído e testado.

Informação sobre a próxima aula

Na Aula 2, serão introduzidos mais conceitos básicos, como variáveis, expressões e leitura de dados, para que você possa fazer programas para lidar com tarefas mais complexas.

Referências bibliográficas

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

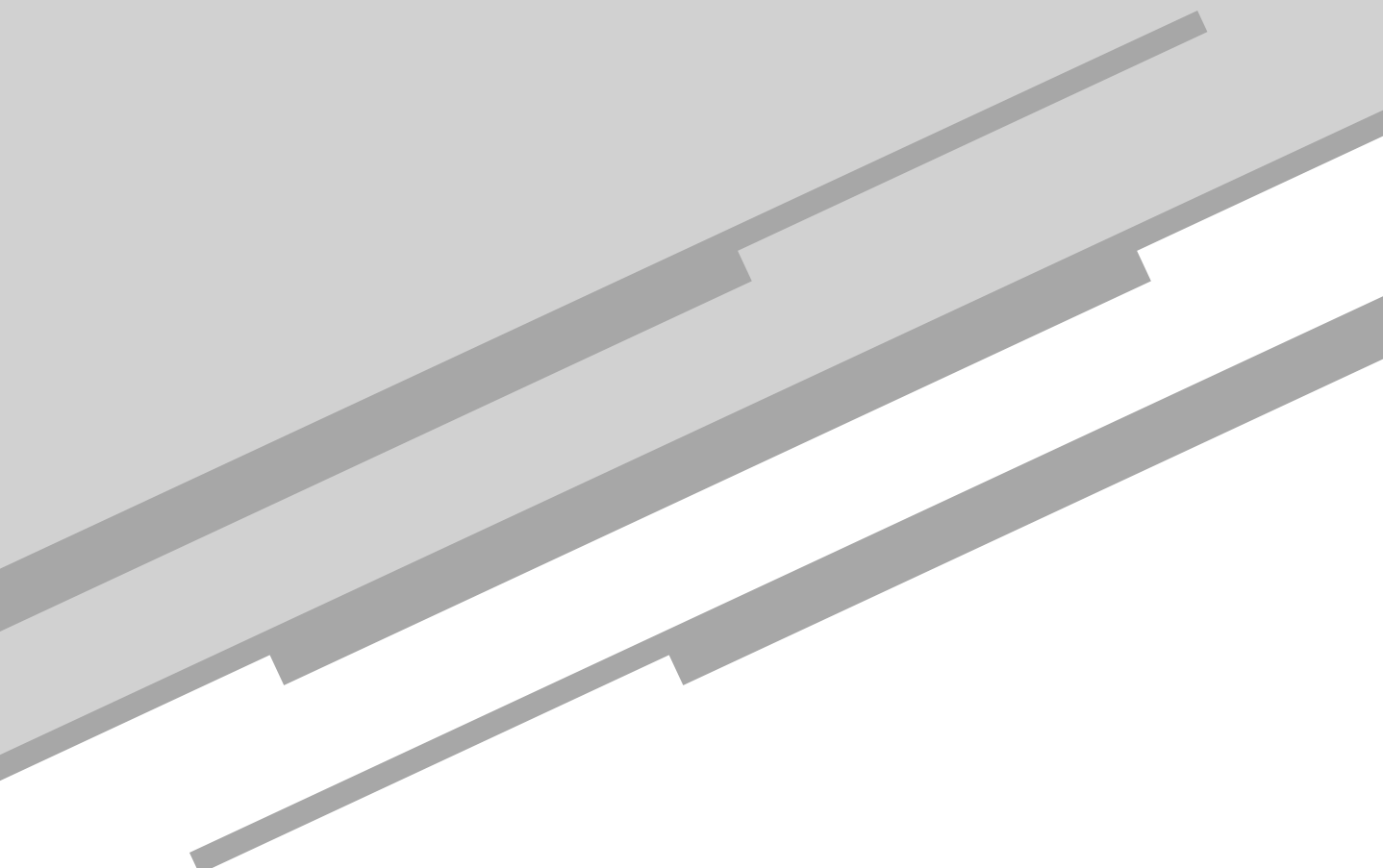
FARRER, H. et al. *Programação estruturada de computadores*. 2ª ed. Rio de Janeiro: Guanabara, 1989.

UBUNTU, 2018. Desenvolvido por Canonical Ltda. Disponível em: <www.ubuntu.com>. Acesso em 26 fev. 2018.

UBUNTUPT. Comunidade Portuguesa, 2010. Desenvolvido por Canonical Ltda. Disponível em: <www.ubuntu-pt.org>. Acesso em: 26 fev. 2018.

Aula 2

Variáveis e entrada/saída de dados



Meta

Expor os conceitos iniciais para criação dos primeiros programas.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. criar programas capazes de fazer operações matemáticas e textuais;
2. utilizar variáveis na construção de um programa;
3. fazer com que os programas interajam com o usuário.

Sintaxe básica

Para construir programas em Java (ou em qualquer outra linguagem de programação), sempre é necessário seguir a sintaxe da linguagem própria para programação desta linguagem (no caso a sintaxe de Java). Sintaxe é o conjunto de regras que dizem se um programa pode ou não ser executado por um computador.

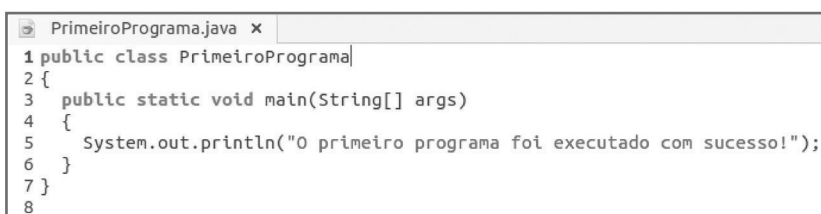
Observe as seguintes frases:

1. “O livro da capa dura é vermelho.”
2. “dura da vermelho é. capa livro O”

É fácil perceber que a frase dois é simplesmente uma permutação dos termos da frase um, porém, a frase dois não tem nenhum sentido. Neste caso, uma simples mudança na ordem dos termos tornou a frase sem sentido. O mesmo pode ocorrer nas linguagens de programação. Pequenas mudanças podem fazer com que os programas não possam ser executados pelo computador. Por isso, você deve sempre ficar atento a erros de sintaxe.

Para exemplificar, vamos começar refazendo um dos programas da Aula 1. O primeiro conceito da sintaxe que você irá aprender é o conceito de comentário.

A **Figura 2.1** mostra um programa java que escreve na tela a seguinte mensagem: “O primeiro programa foi executado com sucesso!”



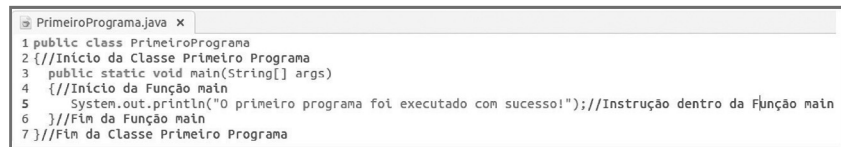
```
1 public class PrimeiroPrograma {
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("O primeiro programa foi executado com sucesso!");
6     }
7 }
8
```

Figura 2.1: Programa inicial para impressão.

Já a **Figura 2.2** mostra o mesmo código, porém com a inserção de comentários.

Foram inseridos, ao todo, cinco comentários:

- dois para marcar o início e fim da classe,
- dois para marcar o início e fim da função e
- um para marcar a instrução a ser executada.



```

1 public class PrimeiroPrograma
2 { //Inicio da Classe Primeiro Programa
3     public static void main(String[] args)
4     { //Inicio da Função main
5         System.out.println("O primeiro programa foi executado com sucesso!"); //Instrução dentro da Função main
6     } //Fim da Função main
7 } //Fim da Classe Primeiro Programa

```

Figura 2.2: Programa inicial para impressão comentado.

Os comentários não são nada mais que texto inserido no arquivo do programa, mas que não fazem parte do programa. Isto significa que, apesar de estarem dentro do código-fonte, estes comentários serão ignorados pelo compilador no momento de construir o programa. Comentários são úteis para inserir informações para outras pessoas que lerão o código ou para colocar informações sobre o próprio código, como, por exemplo, quem foi o autor, data em que foi feito, etc.

Os comentários em Java podem seguir duas sintaxes distintas. A primeira é utilizada para comentários de linha e é expressa da seguinte maneira:

//texto que servirá como comentário

Este tipo de comentário sempre começa com duas barras e não pode ter mais do que uma linha, logo são indicados para informações simples. No exemplo dado na **Figura 2.2**, todos os comentários são de linha.

A segunda sintaxe é utilizada para comentários de blocos, ou seja, para mais do que uma linha. Este tipo de comentário pode ser expresso da seguinte maneira:

/*

texto

que

servirá

como

comentário

***/**

Este tipo de comentário sempre começa com “/*” (barra, asterisco) e termina com “*/” (asterisco, barra). Ele pode compreender várias linhas e é indicado para explicações mais detalhadas do código.

É importante mencionar que é permitido colocar acentuação nos comentários, uma vez que eles não fazem parte do programa. Como os trechos comentados são ignorados pelo compilador, não há problemas em inserir caracteres especiais dentro destes trechos.

Atividade 1

Atende ao objetivo 2

Para verificar que os comentários não influenciam na execução dos programas, use o exemplo que você criou na Aula 1 (**Figura 2.1**) e insira nele um comentário qualquer. Se você utilizar um comentário de bloco, certifique-se de que não colocou nenhuma parte válida do programa dentro do bloco comentado. Compile o programa utilizando o comando “javac” e execute o programa utilizando o comando “java”, exatamente como feito na primeira aula. Qual foi o resultado?

Resposta comentada

Você vai ver que o resultado é o mesmo, com ou sem os comentários. Como o comentário não é parte do programa, o compilador não gera comandos referentes aos comentários, logo o programa será gerado levando em conta somente os trechos não comentados.

As chaves (“{” e “}”) em Java são as marcações de início e fim de um bloco. Um bloco nada mais é do que um conjunto composto por zero ou mais instruções. Como pode ser observado nas **Figuras 2.1** e **2.2**, existem dois pares de chaves, indicando a existência de dois blocos **aninhados**.

As chaves mais externas são as chaves da classe, e as mais internas são as chaves da função *main*. Todo bloco aberto deve ser devidamente fechado, portanto, certifique-se de que todas as chaves abertas têm os pares correspondentes.

Aninhar
no contexto de programação, é o ato de colocar estruturas dentro de outras estruturas. No caso das **Figuras 2.1** e **2.2**, a função está aninhada dentro da classe.

Como dito anteriormente, as estruturas de classe (*public class*) e de função (*public static void main(String[] args)*) serão explicadas em momentos mais oportunos. Por enquanto, trate estas duas partes como sintaxe obrigatória para a construção dos programas Java.

O nome da classe, no entanto, merece uma explicação. Você já aprendeu que o nome da classe tem que ser o mesmo nome do arquivo. Contudo, nem todo nome pode ser utilizado. Dentro da computação, existe um conjunto de regras utilizado para dar nomes para as diversas estruturas, como algoritmos, variáveis, funções, etc. Os nomes utilizados são os chamados **identificadores válidos**.

Identificador válido

É o conjunto de caracteres composto por letras, números e o caractere sublinhado (_). Este conjunto sempre começa com uma letra e não pode conter espaços (DEITEL, 2011).

Um ponto importante sobre identificadores válidos é que eles devem, na medida do possível, expressar claramente a finalidade para a qual serão utilizados. Por exemplo: suponha que você está criando um programa para calcular a raiz quadrada de um número. Qualquer identificador válido pode ser utilizado para dar nome para a classe deste programa, como `a1b2`, `abcd` ou ainda `RaizQuadrada`. Contudo, é considerada uma boa prática, principalmente na computação, escolher identificadores que expressem com clareza a finalidade do algoritmo. Neste caso, o identificador `RaizQuadrada` seria o mais indicado.

É importante ressaltar que Java é sensível às letras capitais (maiúsculas). Assim, os identificadores `RaizQuadrada` e `raizQuadrada` são considerados diferentes. Você deve se certificar de que, onde os nomes devem ser iguais, eles apareçam com as letras maiúsculas nos lugares certos.

O conceito de identificador válido será sempre utilizado para dar nomes às classes, variáveis e funções. Portanto, é necessário sempre ter este conceito em mente.

Atividade 2

Atende ao objetivo 2

Assinale V para verdadeiro ou F para falso, verificando se as seguintes cadeias de caracteres são identificadores válidos ou não:

- | | |
|--------------------------|-----------------------------|
| a) () <code>main</code> | d) () <code>teste_1</code> |
| b) () <code>a1b2</code> | e) () <code>teste 1</code> |
| c) () <code>1a2b</code> | |

Resposta comentada

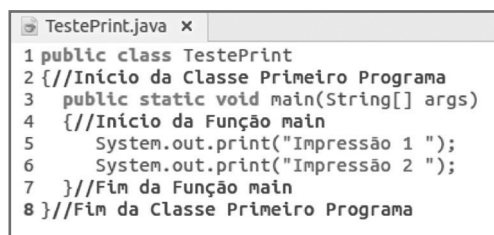
a) V; b) V; c) F; d) V; e) F. A terceira cadeia de caracteres e a última não respeitam as regras de formação de identificadores válidos. A terceira começa com um número, e a última contém espaços. As demais opções satisfazem todas as regras, portanto são identificadores válidos.

Comando de impressão

Todo programa deve produzir pelo menos um resultado. Logo, é necessário um mecanismo para que os programas possam exibir suas saídas. Um destes mecanismos são os comandos de impressão. Este comando é utilizado para enviar mensagens simples para o usuário na forma de texto.

A linguagem Java dispõe de muitos mecanismos para fazer impressão. Dois dos mais comuns são os métodos *println* e *print*. Estes dois métodos podem ser acessados através do objeto *out*, que se encontra dentro da classe *System*. Logo, para acessá-los, é necessário utilizar o nome da classe (*System*), seguido de um ponto (*.*), seguido do nome do objeto (*out*), seguido de outro ponto (*.*), seguido do nome do método (*println* ou *print*).

Um exemplo do uso do *println* já foi dado na **Figura 2.1**, e um exemplo de uso do método *print* é apresentado na **Figura 2.3**.



```
TestePrint.java x
1 public class TestePrint
2 { //Início da Classe Primeiro Programa
3   public static void main(String[] args)
4   { //Início da Função main
5     System.out.print("Impressão 1 ");
6     System.out.print("Impressão 2 ");
7   } //Fim da Função main
8 } //Fim da Classe Primeiro Programa
```

Figura 2.3: Usando o método *print*.

Note que os dois comandos possuem sintaxes de uso muito similares. A única diferença no uso é o nome deles (*print* e *println*). Esta sintaxe de encadeamento através do operador ponto (*.*) significa que estamos fazendo uma chamada ao método (*print*), a partir do objeto

(*out*), que está dentro da classe (*System*). Isto significa que existe, dentro do programa, uma classe *System* que possui um objeto *out* e, a partir deste objeto podemos chamar o método *print*.

Esta sintaxe ficará mais clara quando você estiver estudando criação de objetos e classes. Por enquanto, trate isso como sintaxe obrigatória sempre que quiser imprimir algo.

Os métodos *print* e *println* imprimem a expressão que é passada entre os parênteses. No caso da linha 5 da **Figura 2.3**, a expressão passada foi “Impressão 1”, ou seja, uma expressão de texto. Toda expressão de texto em Java é sempre descrita entre aspas duplas, portanto, sempre que for necessário enviar uma mensagem de texto para o usuário do seu programa, ela deve seguir a seguinte sintaxe: “mensagem”.

Uma diferença entre os métodos *print* e *println* é o modo como eles expõem as expressões. Se você compilar e executar o programa da **Figura 2.3**, verá que o resultado é o seguinte:

```
Impressão 1 Impressão 2
```

Caso você substitua o *print* por *println* nas linhas 5 e 6, o resultado será o seguinte:

```
Impressão1
Impressão 2
```

Como pode ser percebido, o comando *print* exhibe a expressão, mas não muda de linha. Já o comando *println* sempre muda de linha ao terminar a expressão. Portanto, apesar de terem sintaxes muito parecidas os métodos *println* e *print* possuem usos distintos.

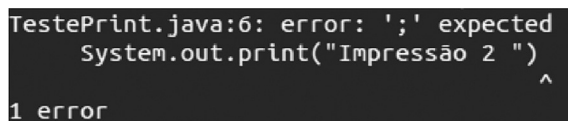
Outra coisa que você deve observar na sintaxe básica é o uso do ponto-e-vírgula (;).

Em Java, sempre que queremos executar uma instrução, ela deve ser seguida de um (;).

Note, por exemplo, a **Figura 2.1**. Neste caso, só há uma instrução e, logo no final dela, deve haver um (;). Já na **Figura 2.3**, há duas instruções (duas chamadas do método *print*), logo deve haver dois (;): um para cada instrução.

As instruções de um programa podem ser de vários tipos: criação de variáveis, atribuição de valores, chamadas de métodos (caso das figuras mostradas até aqui), etc. No entanto, um dos pontos comuns entre as

instruções é o uso do operador (;) no final de uma instrução. Não usar este operador é um erro de sintaxe, e o compilador mostrará uma mensagem similar à exposta na **Figura 2.4** toda vez que você cometer este erro.



```
TestePrint.java:6: error: ';' expected
    System.out.print("Impressão 2 ")
                        ^
1 error
```

Figura 2.4: Erro exibido no terminal quando se esquece um (;) no final da instrução. Note que a mensagem indica o nome do arquivo em que o erro ocorreu, seguido da linha onde o erro está localizado. Neste caso, o erro ocorreu na sexta linha do arquivo “TestePrint.java”.

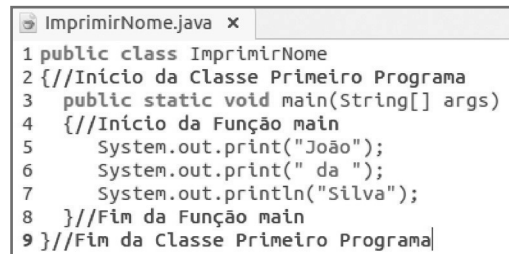
Atividade 3

Atende ao objetivo 3

- Faça um programa java que escreva todas as partes do seu nome (nome, sobrenome) na mesma linha.
- Faça também um segundo programa que escreva cada parte do seu nome em uma linha diferente.

Resposta comentada

- Para fazer estes programas, tudo o que você deve fazer é utilizar corretamente os métodos *print* e *println*. Suponha que o seu nome seja “João da Silva”. Para o primeiro exercício, uma possível solução seria a seguinte:



```

1 public class ImprimirNome
2 { //Início da Classe Primeiro Programa
3     public static void main(String[] args)
4     { //Início da Função main
5         System.out.print("João");
6         System.out.print(" da ");
7         System.out.println("Silva");
8     } //Fim da Função main
9 } //Fim da Classe Primeiro Programa

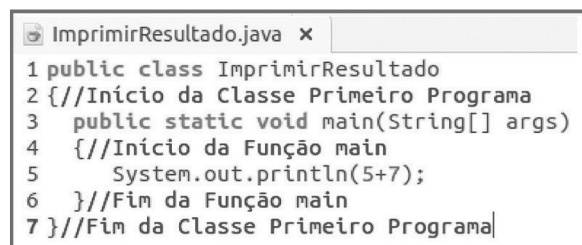
```

Note que esta solução utiliza duas vezes o método *print* e uma vez o método *println*. Como o método *print* não muda de linha, o texto “da” será escrito na mesma linha do texto “João”, logo após este último. De modo análogo, o texto Silva será mostrado logo após o texto “da ”, visto que o método utilizado para imprimir “da” não muda de linha. O último texto é escrito com o método *println*, que muda de linha, mas como não há nada mais a ser escrito, esta mudança de linha não interfere no resultado.

Existem outros modos de se resolver esta questão. Um dos mais simples é utilizar somente uma chamada do método *print* contendo todos os elementos do seu nome na mesma expressão de texto (*System.out.print("João da Silva");*). Ambas as maneiras estão corretas, logo, qualquer uma pode ser considerada como solução deste problema.

b) Para a segunda proposta, utilize a solução da primeira substituindo as chamadas do método *print* por chamadas de *println*.

Outro exemplo de uso dos mecanismos de impressão pode ser visto na **Figura 2.5**. Neste caso, o programa é feito para exibir o resultado final de uma expressão aritmética.



```

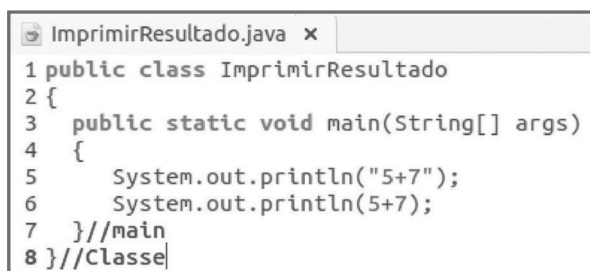
1 public class ImprimirResultado
2 { //Início da Classe Primeiro Programa
3     public static void main(String[] args)
4     { //Início da Função main
5         System.out.println(5+7);
6     } //Fim da Função main
7 } //Fim da Classe Primeiro Programa

```

Figura 2.5: Imprimindo resultado de uma expressão.

No caso da **Figura 2.5**, o que é exibido para o usuário é o número 12, que é o resultado da avaliação da expressão $5 + 7$. O resultado obtido por este programa ocorre porque o comando para imprimir (método *println*) é executado depois que todas as operações forem efetuadas. Normalmente, os comandos são executados da esquerda para a direita (do mesmo modo como escrevemos e lemos uma linha na cultura ocidental), exceto quando há **precedências** diferentes envolvidas. As chamadas de métodos têm precedência menor que as operações aritméticas, logo a soma é executada primeiro e, só depois, o resultado da soma é exibido para o usuário.

Um exemplo interessante do método *println* para diferentes tipos de informação pode ser visto na **Figura 2.6**. Note que, na figura, há duas chamadas do método *println*. Como vimos, algoritmos podem conter mais de uma instrução em seu corpo. Nestes casos, a execução é feita sequencialmente de cima para baixo (exceto em casos específicos, que veremos em outra aula).



```

1 public class ImprimirResultado
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("5+7");
6         System.out.println(5+7);
7     } //main
8 } //Classe

```

Figura 2.6: Imprimindo tipos diferentes de informação.

Na primeira linha, é exibido o texto $5 + 7$, e, na linha subsequente, é exibido o número 12. Como visto, cada chamada de *println* muda de linha após exibir mensagens para o usuário. Portanto, se temos duas chamadas, deveremos ter duas linhas exibidas. Na primeira, é exibido o texto $5 + 7$, porque no programa esta expressão aparece entre aspas, indicando que a expressão a ser impressa é uma expressão de texto. Na segunda linha, a ausência das aspas indica que expressão a ser impressa é o resultado de uma operação, neste caso, aritmética. Assim, a saída para o usuário é mostrada da seguinte maneira:

```

5 + 7
12

```

Precedência (de operações)

Regra que estabelece em que ordem as operações são efetuadas. Um exemplo clássico são as precedências da soma e da multiplicação na matemática. A expressão $2 + 3 \times 5$ tem a operação de multiplicação executada primeiro, apesar de a referida multiplicação estar mais à direita da soma. Isto ocorre porque a precedência da multiplicação é maior que a da soma.

Diversas operações aritméticas podem ser utilizadas no comando imprimir. A **Tabela 2.1** mostra as operações aritméticas, seus respectivos operadores, exemplos de uso, significados e precedências. Quanto maior a precedência de um operador, maior a prioridade para sua execução. Note que vários operadores têm a mesma precedência. No caso de expressões compostas somente por operadores de mesma precedência, a expressão é avaliada da esquerda para a direita.

Tabela 2.1: Precedência de operadores

Operação	Operador	Exemplo de Uso	Significado	Precedência
Inversão de Sinal	-	-2	Menos dois	3
Multiplicação	*	2*3	Dois vezes três	2
Quociente	/	6/2	Quociente de seis dividido por dois	2
Módulo da Divisão	%	4%3	Resto da divisão de quatro por três	2
Soma	+	2+2	Dois mais dois	1
Subtração	-	5-1	Cinco menos um	1



Operação “módulo da divisão” também é chamada de “resto da divisão” ou “modulus”.

Quando fazemos uma divisão de dois números inteiros, obtemos dois resultados: um quociente e um resto. Para acessarmos o QUOCIENTE, usamos o operador (/). Para acessarmos o RESTO, usamos o operador (%). É importante mencionar que o operador MODULUS só pode ser usado para divisão com números inteiros.

A divisão entre dois números reais é sempre um número real; portanto, esta divisão não tem resto, e o operador modulus perde o sentido. Logo, para divisão de números reais, deve-se usar somente o operador de quociente (/).

Para exemplificar, vejamos as seguintes operações: 5/2, 5%2 e 5.0/2.0. A primeira é o quociente da divisão entre dois números inteiros (5 e 2). A segunda é o resto da divisão entre dois números

inteiros (5 e 2). E a última é o quociente da divisão entre dois números reais (5.0 e 2.0). Esta distinção vem da definição de conjuntos numéricos.

No conjunto dos números reais, $5.0/2.0$ resulta em 2,5. Note que os números com casas decimais não se encontram definidos no conjunto dos números inteiros. No conjunto dos números inteiros, a operação de divisão fornece dois resultados, um quociente e um resto. Neste caso, a divisão de 5 por 2 fornece quociente 2 e resto 1. Se quisermos acessar o quociente, utilizamos o operador (/): $5/2$. Se quisermos acessar o resto, utilizamos o operador (%): $5\%2$.

Note que a operação a ser utilizada, depende do resultado que você quer obter. Se você quer dividir 5 reais entre duas pessoas, cada uma leva 2 reais e 50 centavos. Neste caso, uma divisão de números reais faz sentido. Agora, se você quer dividir cinco canetas entre duas pessoas, cada pessoa fica com duas canetas e você tem um excedente de uma caneta. Neste caso, não faz sentido que uma pessoa fique com duas canetas e meia.



Em Java, números reais são expressos com um ponto (.) separando a parte inteira da parte decimal. Além disso, a multiplicação é representada apenas pelo operador (*). Ou seja, a multiplicação de 3 por 4 seria descrita como $3 * 4$ em Java, e não como 3×4 , embora o operador (x) também seja usado em matemática para expressar esta operação.

Em alguns casos, é necessário mudar a ordem de avaliação das expressões aritméticas. Para este fim, utilizam-se os parênteses. Assim, uma expressão do tipo $(2+3)*5$ tem a soma avaliada primeiro.

É importante mencionar que, em Java, diferentemente da matemática, não existem outros operadores para mudar a precedência de avaliação.

A matemática usa, além dos parênteses, colchetes e chaves. Ex: $7 * \{ [(2 + 3) * 5] - 1 \}$. Em Java, como existe apenas o operador parênteses, a expressão anterior ficaria do seguinte modo: $7 * (((2 + 3) * 5) - 1)$. Neste caso, os parênteses mais internos são resolvidos primeiro. Assim, a primeira operação é a soma $2 + 3$. O resultado desta soma é, então, multiplicado por 5. Do resultado da multiplicação, é subtraída uma unidade e, por último, é feita a multiplicação por 7.

===== **Atividade 4** =====

Atende ao objetivo 1

I. Escreva o resultado das seguintes expressões:

- | | |
|-----------------|--------------------|
| a) $1 + 2 * -1$ | c) $1 + 4 * 5 - 2$ |
| b) $1 * 2 / 3$ | d) $1 \% 2 - 1$ |

II. Indique o resultado das seguintes expressões. Considere que somente números inteiros são utilizados:

- | | |
|--------------------|-------------------------|
| a) $2 + 3 * 4$ | d) $2 + 3 / (4 - 1)$ |
| b) $(2 + 3) * 4$ | e) $(4 \% (3 - 2)) * 1$ |
| c) $2 + 3 / 4 - 1$ | f) $4 \% 3 - 2 * 1$ |

III. Coloque parênteses na expressão a seguir, de modo que o resultado seja final seja 71:

$$3 + 6 * 8 - 3 / 7 - 4$$

Resposta comentada

I.

a) -1 . O -1 é avaliado primeiro; em seguida, a multiplicação por 2 e, por último, a soma com 1.

b) 0 . A multiplicação é feita primeiro, resultando em 2. Em seguida, é tomado o quociente da divisão de 2 por 3, que resulta em 0.

c) 19 . A multiplicação é feita primeiro; em seguida, a soma e, por último, a subtração.

d) 0 . É importante mencionar que o operador de resto da divisão só pode ser utilizado para números inteiros, e que a divisão inteira tem dois resultados, o quociente e o resto. Se quisermos o quociente, escrevemos $1 / 2$, Se quisermos o resto, escrevemos $1 \% 2$. A primeira operação resulta em 0, a segunda resulta em 1. Se estes números forem reais, o resultado é um número real. Deste modo, $1.0 / 2.0$ resulta em 0.5. Como o operador (%) tem maior precedência que o operador (-), o resto da divisão é resolvido primeiro, sendo que $1 \% 2$ resulta em 1. Ao subtrairmos 1, obtemos 0 como resultado final.

II. a) 14; b) 20; c) 1; d) 3; e) 0; f) -1.

III. $((3 + 6) * 8) - 3 / (7 - 4)$

Tipo

Quando se trata de variáveis, os tipos expressam a natureza da informação armazenada. Toda variável tem um tipo, e podem existir diversas variáveis do mesmo tipo. Alguns tipos, como números inteiros, números reais e textos, por exemplo, são comuns em praticamente todas as linguagens. A tipagem é importante para que os programadores possam tratar de forma coerente as posições de memória.

Números reais de precisão dupla

Na computação, os números reais são representados na forma de potências de uma base. Como exemplo, o número 35 é representado como 0.35×10^2 , usando a base 10. Números reais em Java seguem o padrão IEEE 754, que especifica que números de precisão simples (chamados de float em Java) usam 32 bits para armazenar as casas decimais e os expoentes. Já os números de precisão dupla (*double*) usam 64 bits para armazenar as casas decimais e o expoente, o que por sua vez implica maior precisão nos cálculos realizados.

Variáveis

Variáveis são posições de armazenamento (memória) associadas a um identificador (nome) que são utilizadas para guardar informações (valores). Em linguagens fortemente tipadas (Java e C++, entre elas), as variáveis são também associadas a um **tipo**, que exprime qual a natureza da informação presente naquela posição de memória (número inteiro, número real, texto, etc.). As sintaxes de declaração de variáveis em Java são as seguintes:

tipo nome;

tipo nome1, nome2, ... , nomen;

A primeira é utilizada para declarar uma única variável de um determinado tipo. A segunda declara várias variáveis de um mesmo tipo. Note que é necessário o uso do “;” no final de ambas as declarações.

Dentre os tipos que são utilizados com maior frequência são:

- *int* (para números inteiros),
- *double* (para **números reais de precisão dupla**),
- *boolean* (para valores **lógicos/booleanos**),
- *String* (para conjuntos de caracteres).



Note que as informações são processadas pelo computador no formato binário (composto de zeros e uns). Como você pode imaginar, o conjunto de possibilidades de representações neste vocabulário é limitado e, portanto, o mesmo conjunto de zeros e uns pode representar várias informações distintas. Os tipos são utilizados para diferenciar estas informações. Para exemplificar, imagine o caracter “A” (letra a maiúscula) e o número 65. Estas duas informações possuem natureza completamente distinta: uma é uma letra, a outra é um número. Entretanto, em Java, elas têm a mesma representação binária. Portanto, a única coisa que nos impede de fazer uma multiplicação de um número qualquer pela letra A é a informação de que A é do tipo texto.

A linguagem Java possui outros tipos de dados como *byte*, *short*, *long*, *float* e *char*, além de uma série de tipos de dados compostos. Porém, estes tipos não serão explicados neste momento.

Deste modo, usando-se os tipos, uma declaração de uma variável inteira para armazenar a idade de uma pessoa em anos pode ser feita da seguinte forma: *int idade*. De modo semelhante, a declaração de duas variáveis reais para expressar o salário de uma pessoa e sua altura em metros pode ser feita do seguinte modo: *double salario*, *altura*.

Na computação, uma variável é uma posição de armazenamento (uma parte da memória RAM), e o valor da informação armazenado nesta posição pode mudar ao longo da execução de um programa. Neste sentido, as variáveis computacionais são semelhantes às variáveis utilizadas em funções na matemática, visto que, nas funções, as variáveis podem assumir diferentes valores.

Para modificar os valores das variáveis, é utilizada a operação de atribuição, para a qual Java utiliza o operador (=). A atribuição pode ser feita no momento da criação de uma variável ou em qualquer outro ponto do programa. Assim, se em alguma parte de um código, escrevermos:

```
int idade = 3;
System.out.println(idade);
```

ou

```
int idade;
idade = 3;
System.out.println(idade);
```

... o resultado será o mesmo. O que será exibido é o valor da informação contido na variável *idade*. Neste caso, 3.



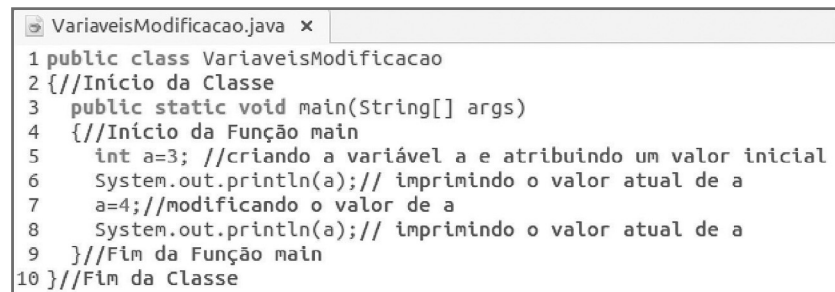
Diferentes linguagens de programação possuem diferentes *tipos* e diferentes *operadores*. É preciso ficar atento com detalhes de sintaxe no momento de transcrever algoritmos em código-fonte. Por exemplo, Java utiliza o operador (=) para atribuições. Para esta mesma operação, a linguagem Pascal utiliza o operador (:=).

Lógico/Booleano

O tipo **lógico**, também chamado de **booleano**, pode assumir dois valores: Verdadeiro (*true*) ou Falso (*false*). São comumente utilizados em estruturas de controle, como desvio e repetição, para verificar a veracidade de certas sentenças. Um exemplo de uso deste tipo é o resultado da expressão $3 > 5$. Neste caso, a expressão resulta em Falso. Este resultado falso pode ser armazenado em variáveis do tipo *boolean* para ser utilizado em outras partes do código. Estas estruturas serão vistas em momentos oportunos.

A **Figura 2.7** mostra um programa no qual o valor de uma variável é modificado.

Na quinta linha do código, é criada uma variável chamada *a*, à qual é atribuído o valor 3. Em seguida, o valor de *a* é exibido. A sétima linha modifica o valor de *a* para 4. Por último, o novo valor de *a* é exibido novamente.



```

1 public class VariaveisModificacao
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         int a=3; //criando a variável a e atribuindo um valor inicial
6         System.out.println(a); // imprimindo o valor atual de a
7         a=4; //modificando o valor de a
8         System.out.println(a); // imprimindo o valor atual de a
9     } //Fim da Função main
10 } //Fim da Classe

```

Figura 2.7: Programa que faz mudança no valor de uma variável.

Assim, o que é exibido pelo programa da **Figura 2.7** são as seguintes linhas:

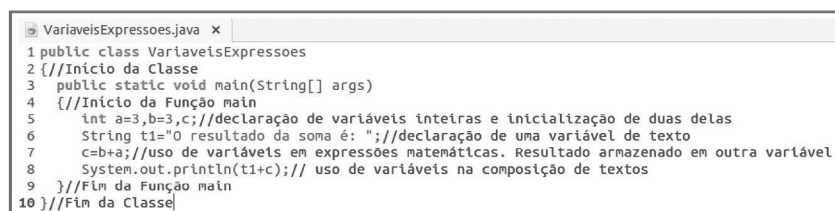
```

3
4

```

Também é possível utilizar variáveis em expressões aritméticas e para compor textos mais complexos.

A **Figura 2.8** mostra um programa que faz uso de variáveis em expressões matemáticas e em construção de textos compostos.



```

1 public class VariaveisExpressoes
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         int a=3,b=3,c=3; //declaração de variáveis inteiras e inicialização de duas delas
6         String ti="O resultado da soma é: "; //declaração de uma variável de texto
7         c=b+a; //uso de variáveis em expressões matemáticas. Resultado armazenado em outra variável
8         System.out.println(ti+c); // uso de variáveis na composição de textos
9     } //Fim da Função main
10 } //Fim da Classe

```

Figura 2.8: Expressões com variáveis numéricas e textuais.

Na quinta linha do código, são declaradas três variáveis inteiras: *a*, *b* e *c*. Para *a* e *b*, são atribuídos valores iniciais.

Na sexta linha, é criada uma variável textual que contém a expressão “O resultado da soma é: ”.

Na sétima linha, é calculada a soma entre os valores contidos nas variáveis *a* e *b*, e o resultado desta soma é armazenado em *c*. Isto ocorre porque o operador de atribuição tem precedência menor que o operador de soma, logo a soma é feita primeiro.

Na linha oito do corpo, está o comando `System.out.println(t1+c);`, que exibe a concatenação do texto contido em *t1* com o valor contido em *c*. Assim, o que é exibido para o usuário é a expressão “O resultado da soma é: 6”.

Note que, apesar de a operação de concatenação também utilizar o operador (+), o que está sendo feito não é a soma entre um texto e um número, e sim a justaposição de um texto com um número.

A operação de concatenação pode ser utilizada tanto entre números e texto quanto entre textos. O fato de um mesmo operador – neste caso, (+) – poder ser utilizado para mais de uma operação se deve à **sobrecarga de operadores**.

Além da operação de concatenação, outra operação comum em textos é o acesso aos caracteres. Por exemplo: suponha que temos uma variável de texto *t1* com a expressão “abcde”. Para termos acesso somente à letra “a”, utilizamos a sintaxe `t1.charAt(0)`. Para acessarmos a letra “b”, utilizamos `t1.charAt(1)`, e assim por diante até `t1.charAt(4)`, que retorna à letra “e”.



Na computação, existem linguagens em que a contagem de caracteres de texto começa em 0 (C++, Java) e em 1 (Pascal). É preciso atenção para adaptar algoritmos para as características da linguagem utilizada.

Uma **String** em Java é um objeto que armazena um conjunto de caracteres. Ao invocar o método `charAt` para um determinado objeto, passando como parâmetro uma posição desejada, recebemos como

Sobrecarga de operadores

Utilizar um operador para mais de uma operação. O operador (-), por exemplo, pode ser utilizado para duas operações, de acordo com a **Tabela 2.1**: a subtração e a inversão de sinal. O contexto indica a qual operação o operador se refere em um trecho de código. Por exemplo, na expressão `-1 - 2`, há dois operadores (-). O primeiro se refere à operação de inversão de sinal, o segundo se refere à operação de subtração.

respostao caractere contido na posição desejada que está armazenado dentro do objeto a partir do qual o método foi invocado.

Observe a **Figura 2.9**, na qual está representada como funciona internamente um objeto *String* que contém a palavra “casa”. No momento em que o texto é associado a uma variável chamada *t* pelo comando de atribuição, é criado um conjunto de posições e cada letra é armazenada em uma posição do conjunto. No fim, *t* serve como nome deste conjunto. As posições são associadas a números, como visto na figura. Assim, a primeira letra fica sempre na posição 0, a segunda letra na posição 1 e assim sucessivamente. Você estudará em mais detalhes este conceito de conjuntos quando estiver estudando matrizes.

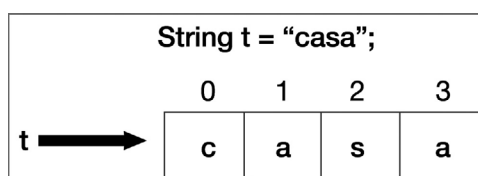


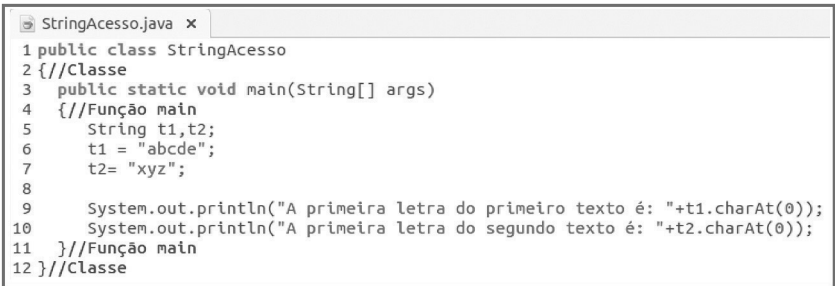
Figura 2.9: Representação de texto em Java.

Neste sentido, podem-se ter múltiplas variáveis do tipo *String*, cada uma contendo um texto diferente, e pode-se chamar o método *charAt* para cada uma delas, sem que uma interfira na execução do outro.

A **Figura 2.10** mostra um programa que faz uso do acesso a caracteres. Neste exemplo, é exibido para o usuário “a primeira letra do primeiro texto é a” e, em seguida, é exibido o texto “a primeira letra do segundo texto é x”.

Note que o método *charAt(0)* é utilizado para as duas variáveis de texto (*t1* e *t2*) e que ele retorna o primeiro caractere de cada uma delas corretamente.

Em cada uma das chamadas, o método *charAt(0)* retorna à primeira letra do texto a partir do qual ele foi invocado. Na linha 9 da **Figura 2.9**, este método é chamado a partir do objeto *t1*, logo ele irá retornar a primeira posição do texto contido em *t1*. Como na linha 10 ele é invocado a partir do objeto *t2*, ele retornará um dos caracteres relativos a *t2*.



```

1 public class StringAcesso
2 { //Classe
3     public static void main(String[] args)
4     { //Função main
5         String t1,t2;
6         t1 = "abcde";
7         t2= "xyz";
8
9         System.out.println("A primeira letra do primeiro texto é: "+t1.charAt(0));
10        System.out.println("A primeira letra do segundo texto é: "+t2.charAt(0));
11    } //Função main
12 } //Classe

```

Figura 2.10: Programa que faz uso do acesso aos caracteres de uma variável do tipo *String*.

Outros métodos relativos a textos que são frequentemente úteis são *length*, que retorna o número de caracteres que um texto possui, e *compareTo*, que compara dois textos.

São diversos os métodos disponíveis para uso na API Java e, por isso, frequentemente será necessário que você consulte a documentação desta API. Esta documentação está disponível no Oracle Help Center e pode ser encontrada em <https://docs.oracle.com/en/java/>. Trata-se de um catálogo que contém as classes disponíveis, os métodos de cada classe, suas descrições e exemplos de uso.

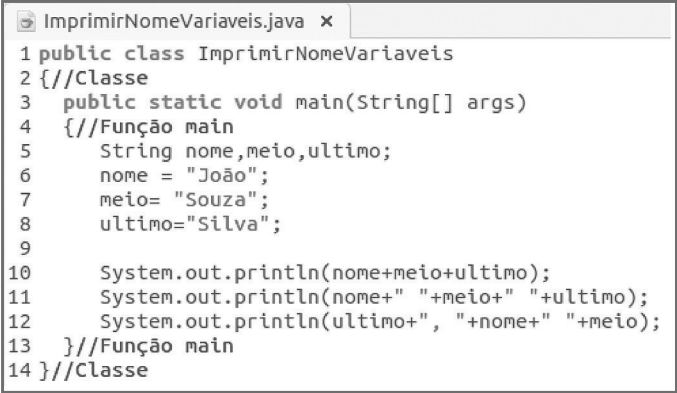
Atividade 4

Atende ao objetivo 1

- Faça um programa para exibir o seu nome;
- Faça um programa que imprima em uma linha o seu nome normalmente e, em outra linha, imprima conforme feito na língua inglesa. Use variáveis de texto. Exemplo: para o nome *João Souza Silva*, o algoritmo deve imprimir em uma linha *João Souza Silva* e, na outra, *Silva, João Souza*.

Resposta comentada

Como a primeira questão é parte da segunda, apenas a solução para a segunda atividade será apresentada:



```
1 public class ImprimirNomeVariaveis
2 { //Classe
3     public static void main(String[] args)
4     { //Função main
5         String nome, meio, ultimo;
6         nome = "João";
7         meio = "Souza";
8         ultimo = "Silva";
9
10        System.out.println(nome+meio+ultimo);
11        System.out.println(nome+" "+meio+" "+ultimo);
12        System.out.println(ultimo+", "+nome+" "+meio);
13    } //Função main
14 } //Classe
```

Figura 2.11: Programa que imprime em uma linha o seu nome normalmente e, em outra linha, imprime conforme feito na língua inglesa.

Neste programa, três variáveis textuais são utilizadas: uma para guardar o nome, uma para guardar o nome do meio e outra, para guardar o último nome. A linha 10 imprime a concatenação das três variáveis. A linha 11 imprime a concatenação das três, intercaladas por espaços. Já a linha 12 imprime o nome como é feito na língua inglesa. A saída deste programa é mostrada a seguir:

```
JoãoSouzaSilva
João Souza Silva
Silva, João Souza
```

Note que a simples justaposição das variáveis (linha 10) provoca uma saída incompatível com a solução desejada, portanto é necessário intercalar estas variáveis com espaços em branco (linha 11).

Comando de leitura

Muitas vezes, é necessário interagir com os algoritmos, dando a eles dados para sua execução. Um exemplo comum disso é o fornecimento do endereço de um site que queremos visitar, para o navegador. Para este fim, utilizam-se comandos de entrada ou de leitura. A linguagem Java possui vários modos de leitura. Os primeiros que veremos são os métodos *next* da classe *Scanner*. Os principais métodos que utilizaremos serão:

- *next*, que retorna um texto (*String*) lido;
- *nextInt*, que retorna um número inteiro (*int*);
- *nextDouble*, que retorna um número decimal (*double*);
- *nextBoolean*, que retorna um valor lógico (*boolean*).

Para usar estes métodos, é necessária a criação de um objeto do tipo *Scanner*. A linguagem Java possui uma infinidade de classes prontas para o uso, sendo a classe *Scanner* uma delas.

Quando utilizamos classes que não estamos definindo no arquivo corrente, é necessário informar ao compilador onde encontrar estas classes. Isso é feito através do comando *import*, que deve ser usado com a seguinte sintaxe:

```
import endereço;
```

Isto faz com que o compilador procure a classe que se deseja utilizar no endereço passado. Para a classe *Scanner* especificamente, o endereço é *java.util.Scanner*.

O comando *import* deve ser usado no começo do arquivo, antes da definição da classe. Logo, para utilizar a classe *Scanner*, devemos colocar a seguinte linha antes da definição da nossa classe:

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

A sintaxe de criação de objetos é a mesma utilizada para a criação de variáveis, ou seja, *tipo nome*. Neste sentido, para criar um objeto do tipo *Scanner* chamado “entrada”, basta usar a o seguinte comando:

```
Scanner entrada = new Scanner(<fonte>);
```

A inicialização de um objeto, como pode ser observada, é um pouco diferente da inicialização de uma variável.

A inicialização de uma variável é feita através do operador de atribuição (=), seguido de um valor.

A inicialização de um objeto é feita através do operador de atribuição, seguido de uma chamada de construtor. O construtor da classe *Scanner* pede que seja informada uma fonte. Os dados podem ser lidos de diversas fontes nos computadores: teclado, arquivos, Internet, etc. Para ligar o objeto *entrada* ao teclado, basta substituímos a fonte na chamada do construtor por *System.in*, ficando assim a criação do objeto *entrada*:

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

Esta linha significa que está sendo criado um objeto chamado *entrada*, do tipo *Scanner*, ligado ao teclado. Logo, este objeto é responsável por tratar as informações que serão digitadas.

Questões como a criação de objetos e o uso de construtores, entre outras, serão exploradas em mais detalhes em momento oportuno. Por enquanto, trate o *new Scanner (System.in);* como sintaxe obrigatória.

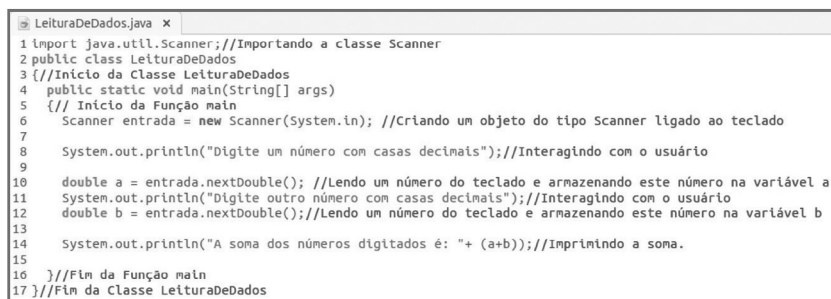
Agora que você já sabe criar um objeto do tipo *Scanner*, basta usar os métodos associados a ele: *next()*, *nextInt()*, *nextDouble()*, etc, para armazenar em variáveis os valores digitados no teclado.

Estes métodos permitem que o usuário digite valores e que o resultado desta digitação seja convertido para um tipo de dado da linguagem Java.

O método *nextDouble()*, por exemplo, converte o valor digitado para um número do tipo *double*. O resultado desta conversão pode ser armazenado em uma variável através do operador de atribuição.

É importante mencionar que estes métodos são bloqueantes, ou seja, enquanto o usuário não inserir uma informação, o programa não prossegue sua execução.

A **Figura 2.12** mostra um exemplo de uso do método *nextDouble()*. Neste programa, dois valores são lidos do teclado e armazenados nas variáveis *a* e *b*. Em seguida, é calculada e exibida a soma dos valores digitados:



```

1 import java.util.Scanner; //Importando a classe Scanner
2 public class LeituraDeDados
3 { //Inicio da Classe LeituraDeDados
4     public static void main(String[] args)
5     { // Inicio da Função main
6         Scanner entrada = new Scanner(System.in); //Criando um objeto do tipo Scanner ligado ao teclado
7
8         System.out.println("Digite um número com casas decimais");//Interagindo com o usuário
9
10        double a = entrada.nextDouble(); //Lendo um número do teclado e armazenando este número na variável a
11        System.out.println("Digite outro número com casas decimais");//Interagindo com o usuário
12        double b = entrada.nextDouble();//Lendo um número do teclado e armazenando este número na variável b
13
14        System.out.println("A soma dos números digitados é: "+ (a+b));//Imprimindo a soma.
15    }
16 } //Fim da Função main
17 } //Fim da Classe LeituraDeDados

```

Figura 2.12: Exemplo de uso do método *nextDouble()*.

Note que, a princípio, não é possível determinar o que vai ser exibido pelo programa da **Figura 2.12**. Neste caso, a resposta exibida depende de quais informações serão inseridas pelo usuário.

Caso sejam inseridos, para *a* e *b*, os valores 2 e 3, respectivamente, o algoritmo irá exibir como saída o valor 5. Caso sejam fornecidos os valores 4.1 e 8.2, será exibido o valor 12.3, e assim sucessivamente.

===== **Atividade final** =====

Atende aos objetivos 1, 2 e 3

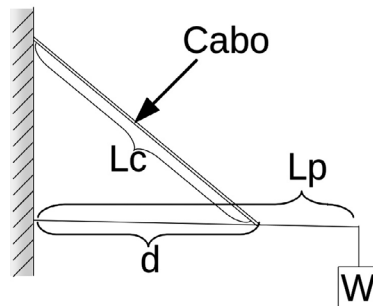
- a) Faça um programa que leia um número inteiro *x* e imprima o oposto dele, ou seja, $-x$.

b) A tensão em um cabo que suporta um peso é dada pela equação a seguir, onde:

T é tensão;

w, o peso;

d, Lp e Lc são distâncias.

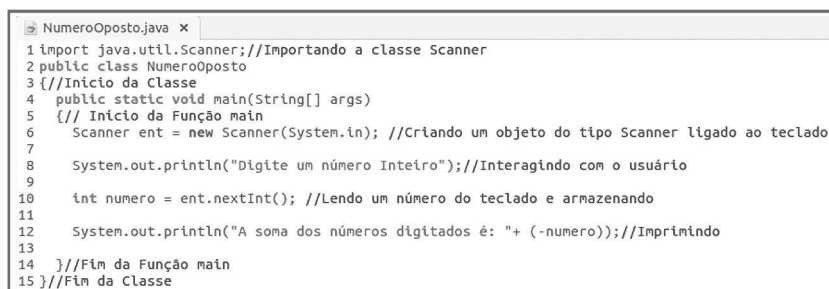


$$T = \frac{W \cdot Lp \cdot Lc}{d \cdot \sqrt{Lc^2 - d^2}}$$

A figura mostra como posicionar as distâncias e o peso. Faça um programa que leia todos os dados necessários e calcule a tensão no cabo. Suponha que os valores digitados serão sempre válidos, ou seja, nenhum valor negativo será passado, nem catetos maiores que hipotenusas etc. Obs: Em Java, a raiz quadrada de um número x é dada pela função *Math.sqrt(x)*.

Resposta comentada

a) O modo mais simples de resolver esta questão é armazenar o valor lido em uma variável e fazer a inversão no próprio comando de impressão. Uma possível resposta para o problema é apresentada a seguir:



```

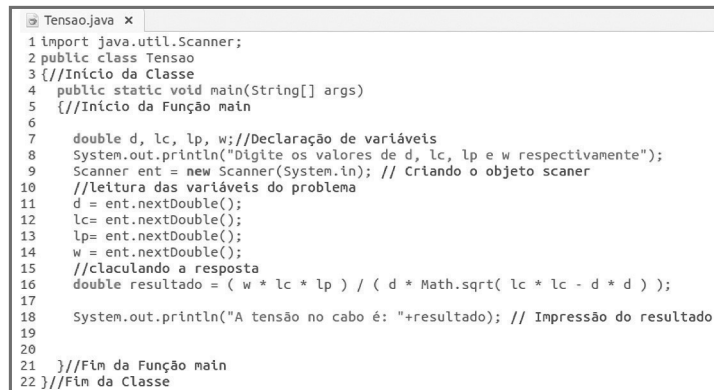
1 import java.util.Scanner; //Importando a classe Scanner
2 public class NumeroOposto
3 { //Início da Classe
4     public static void main(String[] args)
5     { // Início da Função main
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner ligado ao teclado
7
8         System.out.println("Digite um número Inteiro"); //Interagindo com o usuário
9
10        int numero = ent.nextInt(); //Lendo um número do teclado e armazenando
11
12        System.out.println("A soma dos números digitados é: "+ (-numero)); //Imprimindo
13
14    } //Fim da Função main
15 } //Fim da Classe
  
```

Figura 2.13: Solução possível, a partir do armazenamento do valor lido em uma variável, seguido de sua inversão no próprio comando de impressão.

Note que há dois operadores na expressão passada para o método *println*: o operador de concatenação (+) e o operador de inversão de sinal (-). Neste caso, os parênteses são utilizados apenas para explicitar a ordem das operações. Caso você retire os parênteses que envolvem o *-numero*, o resultado permanece o mesmo devido ao fato de as expressões aritméticas terem precedência maior que a concatenação. Portanto, neste caso, primeiro se faz a inversão de sinal, em seguida a concatenação e, por último, o resultado da concatenação é impresso.

b) Apesar de a informação dada sobre valores negativos e a relação entre as distâncias ser óbvia, muitas vezes, o tratamento destas questões acaba por roubar o foco dos programadores. De fato, um programador que trabalha em produção de *software* passa mais tempo tentando impedir o usuário de fazer coisas erradas do que programando o núcleo do *software*. Como o objetivo desta disciplina é introduzir você no mundo da programação, as técnicas usadas para evitar estes erros não serão abordadas. Para as atividades desta disciplina, você pode assumir que o usuário dos programas, que em muitos casos será você, não fará nada que seja inválido.

No que tange à resolução do exercício, a única dificuldade que existe é transformar a equação da tensão da forma matemática, como ela foi apresentada, para uma forma computacional. Uma das possíveis soluções para este problema seria o seguinte programa:



```

1 import java.util.Scanner;
2 public class Tensao
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6
7         double d, lc, lp, w; //Declaração de variáveis
8         System.out.println("Digite os valores de d, lc, lp e w respectivamente");
9         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
10        //leitura das variáveis do problema
11        d = ent.nextDouble();
12        lc = ent.nextDouble();
13        lp = ent.nextDouble();
14        w = ent.nextDouble();
15        //calculando a resposta
16        double resultado = ( w * lc * lp ) / ( d * Math.sqrt( lc * lc - d * d ) );
17
18        System.out.println("A tensão no cabo é: "+resultado); // Impressão do resultado
19
20    }
21 } //Fim da Função main
22 } //Fim da Classe

```

Figura 2.14: Uma das possíveis soluções para a questão.

Note que a expressão é escrita linearmente, tendo o denominador e o numerador delimitados por parênteses. As potências da expressão também foram substituídas por multiplicações simples. Mais adiante, veremos exemplos onde serão utilizadas funções para cálculo de potências de maneira mais natural.

Resumo

Nesta aula, você aprendeu a sintaxe básica de programas Java, bem como a inserir comentários e a lidar com a precedência de operações. Também aprendeu o conceito de variável, como declarar uma variável e como alterar seus valores. Por fim, aprendeu a utilizar os comandos de leitura e escrita, que possibilitam escrever algoritmos que interagem com o usuário.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com situações em que devemos executar trechos de programas dependendo de certos resultados e situações. Para isso, serão introduzidas as estruturas de desvio de fluxo.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. Fundamentos da programação de computadores. São Paulo: Pearson, 2012.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

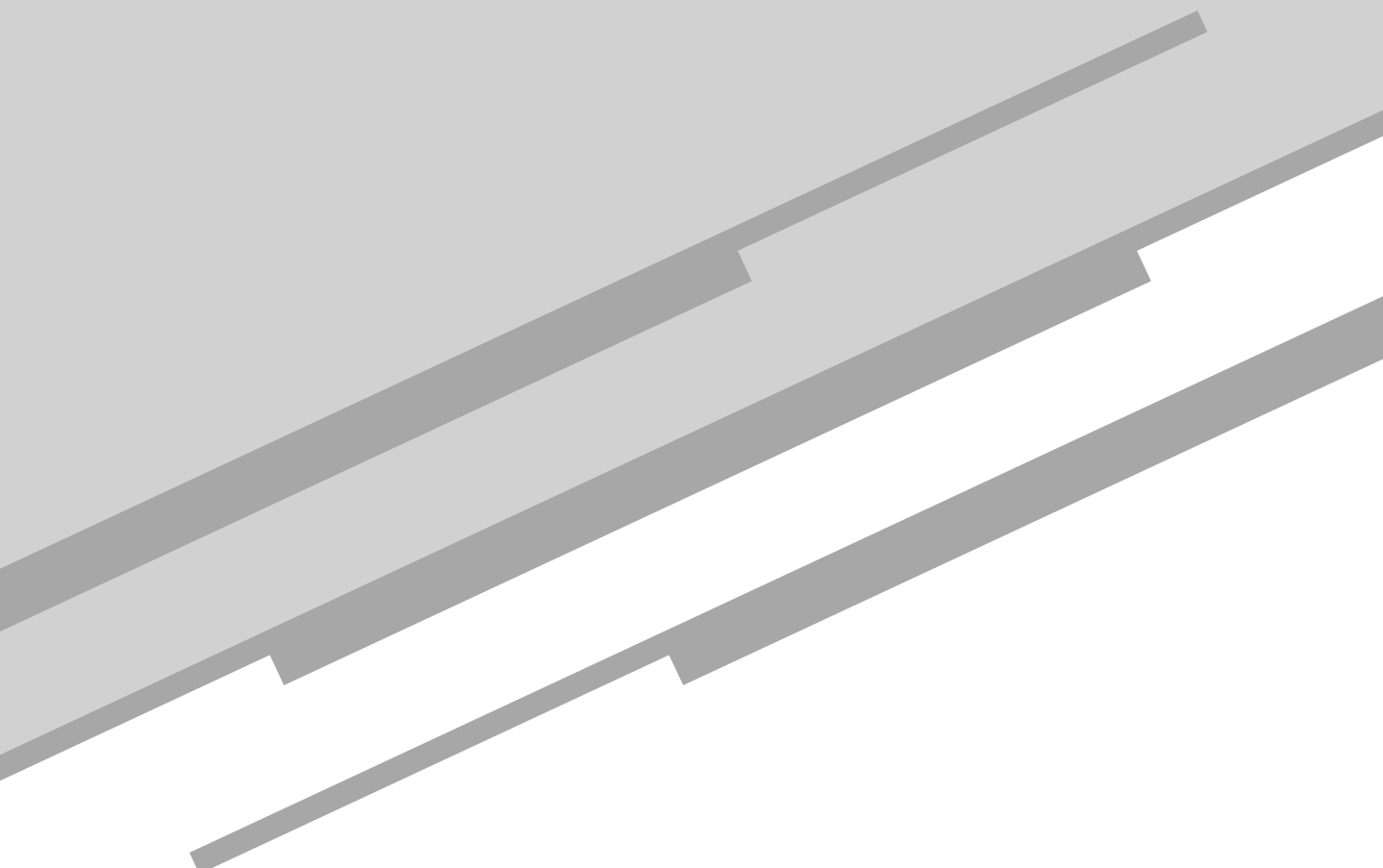
DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. et al. *Programação estruturada de computadores*. 2^a ed. Rio de Janeiro: Guanabara, 1989.

ORACLE HELP CENTER. Java Documentation. Java Platform, Standard Edition (Java SE). Disponível em: <<https://docs.oracle.com/en/java/>>. Acesso em: 02 mar. 2018.

Aula 3

Desvio condicional – Parte I



Meta

Expor os conceitos de desvio condicional e as estruturas usadas para este fim.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. identificar o conceito e o uso de variáveis e expressões lógicas simples;
2. fazer algoritmos que utilizem expressões lógicas simples e desvios condicionais.

Introdução

Muitas vezes, necessitamos executar certas ações em função de algumas condições. Alguns exemplos do cotidiano: devemos lavar roupa, caso exista roupa suja; levamos o guarda-chuva, se o dia está chuvoso; usamos casaco, se está frio; se temos café, tomamos, senão bebemos chá.

Essa necessidade de executar ações na presença de certas condições também existe dentro da computação, como você viu na disciplina de Computação I. Para se analisarem tais condições, são utilizadas estruturas condicionais, também chamadas de estruturas de desvio de fluxo. O uso destas estruturas é frequente e seu domínio é indispensável para a construção de algoritmos. Nesta aula, você verá vários tipos de estruturas de desvio que Java possui. Mas primeiro é preciso dominar o uso de expressões lógicas.

Variáveis e expressões lógicas

Uma *expressão lógica* nada mais é do que um conjunto de operações que pode resultar em dois possíveis valores: *verdadeiro* ou *falso*. Exemplos de expressões lógicas simples são expressões que usam os operadores relacionais como o operador > (maior que).

A **Tabela 3.1** mostra os operadores relacionais e os operadores de igualdade como comumente vistos na matemática, seus correspondentes em Java, exemplos de uso e seus significados.

Operador matemático	Operador em Java	Exemplo	Significado
=	==	a==b	a é igual a b?
≠	!=	a!=b	a é diferente de b?
>	>	a>b	a é maior que b?
<	<	a<b	a é menor que b?
≥	>=	a>=b	a é maior ou igual a b?
≤	<=	a<=b	a é menor ou igual a b?

Tabela 3.1: Operadores matemáticos e seus correspondentes em Java

Um exemplo de avaliação de um operador relacional pode ser $a < b$, tendo a variável a o valor 2, e a variável b , o valor 5. Neste caso, a expressão é traduzida para $2 < 5$ (dois menor que cinco), que resulta em *verdadeiro*. Agora, se mantivermos os valores das variáveis e mudarmos o operador para $>$, temos a expressão $a > b$, que é traduzida para $2 > 5$ (dois maior que cinco), que resulta em *falso*. Em Java, verdadeiro e falso são constantes expressas por *true* e *false*, respectivamente.

É importante mencionar que os resultados de operações relacionais e de igualdade podem ser armazenados em variáveis do tipo lógicas, que em Java são expressas pelo tipo *boolean*. Isto é ilustrado pelo programa da **Figura 3.1**, que irá imprimir *true* ou *false*, dependendo dos valores digitados para as variáveis.

Além disso, é igualmente importante dizer que variáveis do tipo *boolean* só podem assumir dois valores: *true* ou *false*.

```

VariaveisLogicas.java x
1 import java.util.Scanner;
2 public class VariaveisLogicas
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int a,b; //declaração de variáveis inteiras
7         System.out.println("Digite dois números inteiros");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         a = ent.nextInt(); //lendo um número e armazenando em a
10        b = ent.nextInt(); //lendo um número e armazenando em b
11        boolean resposta = a>b; //criando uma variável lógica que armazena o resultado da expressão
12        System.out.println("o resultado da expressão é :"+resposta); // imprimindo o resultado
13    } //Fim da Função main
14 } //Fim da Classe

```

Figura 3.1: Utilização de variáveis lógicas para armazenar resultados de expressões relacionais.



Variáveis do tipo *boolean* são variáveis do tipo lógico (mencionadas anteriormente) e podem assumir dois valores: verdadeiro (*true*) ou falso (*false*). São comumente utilizadas em estruturas de controle, como desvio e repetição, para verificar a veracidade de certas sentenças.

Os operadores relacionais e de igualdade podem ser utilizados com expressões mais elaboradas como, por exemplo, $2 + 2 > 5 - 0$. Neste caso, as expressões aritméticas são resolvidas primeiramente e, somente então, são avaliados os operadores relacionais e de igualdade. Isto ocorre devido à precedência desses operadores.

Atividade 1

Atende ao objetivo 1

Avalie o resultado das seguintes expressões java como *true* ou *false*:

- a) sendo $a = 2$ e $b = 3$, a expressão $a \% b \% a != b \% a$
- b) sendo $a = 3$ e $b = 1$, a expressão $-1 * a - b == -1 * (b - a)$
- c) sendo $a = 1$ e $b = 2$, a expressão $a * b != b * a$

Resposta comentada

- a) Analisaremos a primeira expressão. Como as expressões são resolvidas primeiro, e sempre da esquerda para a direita, $2 \% 3 \% 2$ é resolvida primeiro. Como o resto da divisão de 2 por 3 é 2, este resultado é usado na segunda operação. Logo, $2 \% 3 \% 2$ é parcialmente resolvido como $2 \% 2$. Esta segunda parte da expressão resulta em 0, então, temos $0 != 3 \% 2$. Em seguida, é resolvido $3 \% 2$, que resulta em 1. Assim, a expressão fica $0 != 1$. Como 0 é diferente de 1, a expressão resulta em *true*.
- b) De maneira análoga, a segunda expressão resulta em *false*, c) e a terceira em *false*.

Estrutura de desvio simples

A estrutura de desvio simples, também chamada de “estrutura *if*”, é utilizada quando devemos executar certas ações, caso uma condição seja atendida. Esta instrução é equivalente à estrutura *Se-Então*, que você aprendeu na disciplina de Computação I. Esta estrutura tem duas possíveis sintaxes:

```
if(<expressão Lógica>
{
    <conjunto de instruções>
}

ou

if(<expressão Lógica>
<uma única instrução>
```

Figura 3.2: Sintaxes da estrutura *if*.

A **Figura 3.2** mostra as sintaxes da estrutura *if*. Como sintaxe obrigatória da estrutura, temos: *if*, (e). A palavra *if* indica o início da instrução condicional. O símbolo “(” marca o início da expressão lógica que mostrará a condição que queremos avaliar. O símbolo “)” marca o fim da expressão lógica. A estrutura *if*, em Java, pode ser seguida, ou não, por um bloco de comandos. O bloco sempre inicia com “{” e sempre termina com “}”. Caso seja seguida de um bloco, todas as instruções do bloco serão executadas, se a expressão lógica resultar em *true*. Porém, se a expressão resultar em *false*, todas as instruções do bloco serão ignoradas. Caso a estrutura não seja seguida de um bloco, apenas a próxima instrução será executada ou ignorada, dependendo do resultado da expressão.

A **Figura 3.3** mostra um exemplo de uso da sintaxe com bloco. As instruções antes do *if* são executadas normalmente (de cima para baixo).

Dentro do bloco do *if*, existem três instruções, que serão executadas somente se *a* for maior que *b*. Após o *if*, existe uma instrução que será executada por último.

```

DesvioComBloco.java x
1 import java.util.Scanner;
2 public class DesvioComBloco
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int a,b; //declaração de variáveis inteiras
7         System.out.println("Digite dois números inteiros");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         a = ent.nextInt(); //lendo um número e armazenando em a
10        b = ent.nextInt(); //lendo um número e armazenando em b
11        if(a>b)
12        {
13            System.out.println("o primeiro é maior que o segundo");// executado se a>b
14            System.out.println("o primeiro número é: "+a);// executado se a>b
15            System.out.println("o segundo número é: "+b);// executado se a>b
16        }
17        System.out.println("fim do programa");// mensagem
18    } //Fim da Função main
19 } //Fim da Classe

```

Figura 3.3: Uso de *if* com bloco.

Já a **Figura 3.4** mostra o uso de *if* sem bloco. Os dois programas são muito parecidos, contudo, os resultados são distintos. Neste programa, o *if* se aplica apenas para a instrução da linha 12. Todas as demais instruções são executadas independentemente do resultado da expressão. Note que, se você precisa executar somente uma instrução, caso a condição seja verdadeira, você pode usar qualquer uma das sintaxes. Agora, se você necessita executar mais de uma instrução, você deve optar pelo uso do bloco.

```

DesvioSemBloco.java x
1 import java.util.Scanner;
2 public class DesvioSemBloco
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int a,b; //declaração de variáveis inteiras
7         System.out.println("Digite dois números inteiros");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         a = ent.nextInt(); //lendo um número e armazenando em a
10        b = ent.nextInt(); //lendo um número e armazenando em b
11        if(a>b)
12            System.out.println("o primeiro é maior que o segundo");// executado se a>b
13
14        System.out.println("o primeiro número é: "+a);//mensagem
15        System.out.println("o segundo número é: "+b);//mensagem
16
17        System.out.println("fim do programa");// mensagem
18    } //Fim da Função main
19 } //Fim da Classe

```

Figura 3.4: Uso de *if* sem bloco.

Atividade 2

Atende aos objetivos 1 e 2

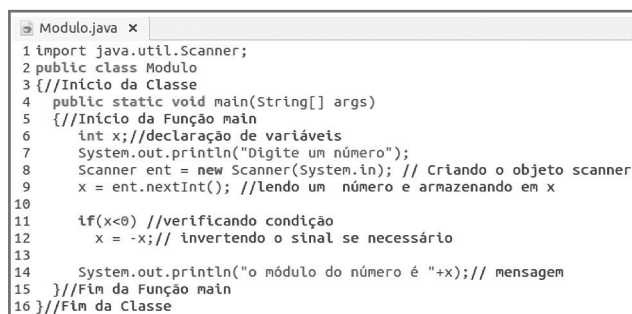
- a) Faça um programa que calcule o valor absoluto de um número inteiro. A equação do valor absoluto é descrita abaixo:

$$|x| = \begin{cases} x, & x \geq 0 \\ -x, & c.c. \end{cases}$$

- b) Faça um programa que leia um número inteiro e verifique se tal número pertence ao conjunto dos números naturais.
- c) Faça um programa que verifique se um determinado número x é múltiplo de outro número y . Os valores de x e y devem ser informados pelo usuário.
- d) Faça um programa que verifique se um determinado número, informado pelo usuário, é múltiplo de 2.

Resposta comentada

- a) De acordo com a equação, o valor absoluto de um número x , também chamado módulo de x , é o próprio x , se x for maior ou igual a zero, ou $-x$, em caso contrário. Desse modo, devemos fazer um programa que imprima o valor de x multiplicado por -1 , caso x seja menor que zero. Uma possível solução para este problema é a seguinte:



```

Modulo.java x
1 import java.util.Scanner;
2 public class Modulo
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int x; //declaração de variáveis
7         System.out.println("Digite um número");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         x = ent.nextInt(); //lendo um número e armazenando em x
10
11         if(x<0) //verificando condição
12             x = -x; // invertendo o sinal se necessário
13
14         System.out.println("o módulo do número é "+x); // mensagem
15     } //Fim da Função main
16 } //Fim da Classe

```

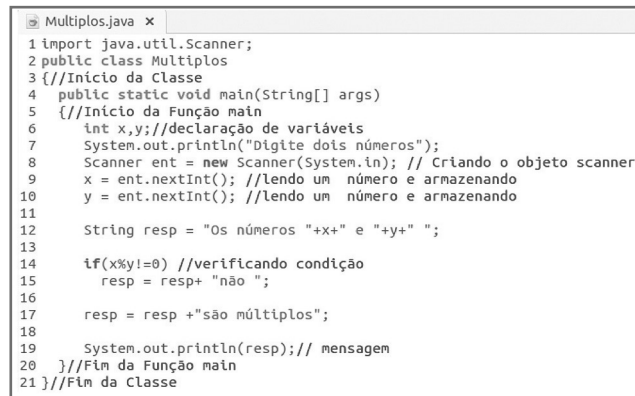
Figura 3.5: Neste programa, a variável x é utilizada tanto para leitura do número quanto para o resultado do valor absoluto.

O primeiro passo é ler o número do qual se deve calcular o módulo. O passo crítico deste algoritmo é a estrutura de desvio. Neste caso, se x for maior ou igual a zero, nada precisa ser feito, pois em x já está o valor absoluto de x (x é positivo), o que atende à equação.

Caso x seja menor que zero, o valor contido em x é multiplicado por -1 , e o resultado é armazenado na própria variável x ($x = -x$). Isso ocorre porque o operador de inversão de sinal tem precedência maior que o de atribuição, logo o lado direito da expressão é avaliado e, só depois, a operação de atribuição é executada. Note que isto atende à definição da equação, uma vez que, se x é menor que zero, ao invertermos o sinal, encontramos o valor absoluto de x .

b) O conjunto dos números naturais está contido no conjunto dos números inteiros. Assim, um número natural necessariamente é inteiro, mas o contrário não é verdadeiro. Os números naturais são compostos de todos os números inteiros positivos mais o zero. Assim, para verificar se um número inteiro pertence ao conjunto dos números naturais, basta verificar se ele é maior ou igual a zero.

Para verificar se um número x é múltiplo de outro número y , deve-se verificar se a divisão de x por y é exata, ou seja, se tem resto igual a zero. Para uma possível solução, primeiramente são criadas duas variáveis com a mesma nomenclatura do problema. A seguir, estas variáveis são lidas, e o resto da divisão entre elas é analisado. Caso o resto seja zero, isto implica que x é um múltiplo de y . Caso contrário, x não é múltiplo de y . A resposta pode ser feita através de construção de texto ou usando dois *ifs* (um para cada caso).



```

1 import java.util.Scanner;
2 public class Multiplos
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int x,y; //declaração de variáveis
7         System.out.println("Digite dois números");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         x = ent.nextInt(); //lendo um número e armazenando
10        y = ent.nextInt(); //lendo um número e armazenando
11
12        String resp = "Os números "+x+" e "+y+" ";
13
14        if(x%y!=0) //verificando condição
15            resp = resp+ "não ";
16
17        resp = resp +"são múltiplos";
18
19        System.out.println(resp); // mensagem
20    } //Fim da Função main
21 } //Fim da Classe

```

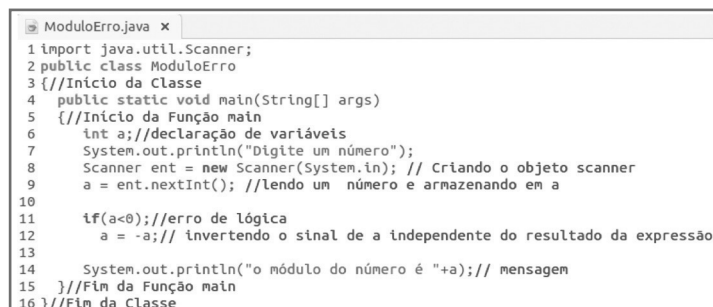
Figura 3.6: A imagem mostra um programa que faz uso de construção de texto.

Nesta solução, um texto é montado para dar a resposta. Exemplo: se os números são 4 e 2, o texto “Os números 4 e 2 são múltiplos” é exibido. Caso os números não fossem múltiplos, um “não” seria colocado para indicar.

d) Como este item é um caso particular do anterior, não há necessidade de repetirmos aqui o desenvolvimento da resposta.

É importante ressaltar que a estrutura *if* não possui “;” (ponto-e-vírgula) em sua sintaxe. Isso ocorre porque o “;” marca o fim de uma instrução. Logo uma fonte frequente de erros é o uso da estrutura *if* (com ou sem bloco) seguido de “;”.

Como você pode notar pelas atividades, a posição da primeira instrução do *if* pode ser colocada na mesma linha ou após linhas em branco. O *if* considera como “sua instrução” aquela que está antes do primeiro “;” após sua colocação. Veja programa da **Figura 3.7**.



```

1 import java.util.Scanner;
2 public class ModuloErro
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int a; //declaração de variáveis
7         System.out.println("Digite um número");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         a = ent.nextInt(); //lendo um número e armazenando em a
10
11        if(a<0); //erro de lógica
12            a = -a; // invertendo o sinal de a independente do resultado da expressão
13
14        System.out.println("o módulo do número é "+a); // mensagem
15    } //Fim da Função main
16 } //Fim da Classe

```

Figura 3.7: Mau uso do “;”, juntamente com a estrutura *if*.

Na linha 11, há um *if*, seguido de um “;”. Neste caso o *if* considera como sendo sua instrução somente aquilo que precede o primeiro “;” depois dele, ou seja a instrução antes do “;” da linha 11. Note que esta é uma instrução vazia, mas ainda assim é uma instrução. Portanto, neste caso, o que vai ser executado, caso a condição seja verdadeira, é uma instrução vazia, o que faz com que a linha 12, seja executada independentemente do resultado da expressão lógica, fazendo assim um programa que não funciona como desejado.

O mesmo pode acontecer caso você esteja utilizando o *if* com bloco. Isso ocorre porque Java permite que você use blocos sem estruturas (isto será visto em mais detalhes em momento oportuno).

Portanto, se o seu programa não está funcionando como deveria, veja se você não posicionou “;” em locais indesejados.

Estrutura de desvio composto

Em muitos casos, devemos tomar cursos de ação diferentes, de acordo com uma condição. Um exemplo do cotidiano é decidir como pagar o restaurante na hora do almoço. Se nós temos dinheiro suficiente à mão, pagamos em dinheiro; senão, pagamos com cartão.

A estrutura condicional composta, também chamada de estrutura *if-else*, é utilizada para resolver problemas deste tipo. Esta estrutura é equivalente a estrutura *Se-Senão*, que você aprendeu em Computação I.

As possíveis sintaxes da estrutura composta são mostradas na **Figura 3.8**.

```
if(<condição>)
    Instrução I
else
    Instrução II

ou

if(<condição>)
{
    Conjunto de Instruções I
}
else
{
    Conjunto de Instruções II
}
```

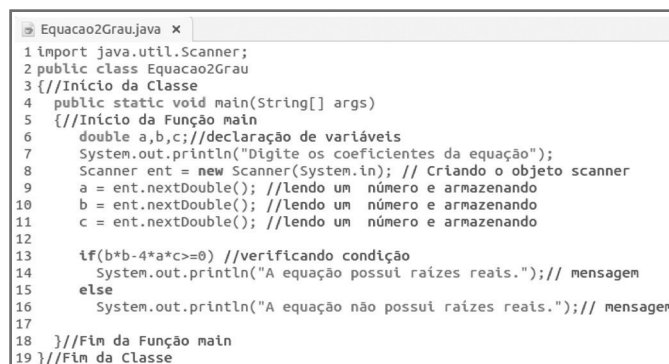
Figura 3.8: Sintaxe da estrutura *if-else*.

Note que a estrutura *if* e a estrutura *if-else* têm muitos elementos em comum. A diferença é que, depois da instrução (ou conjunto de instruções) do *if*, existe a palavra reservada *else* e uma outra instrução (ou conjunto de instruções).

A semântica da estrutura *if-else* é um pouco mais elaborada que a da estrutura *if*. Nesta estrutura, caso a expressão lógica resulte em *true*, o conjunto de instruções I (entre o *if* e o *else*) é executado, e o conjunto de instruções II (posicionadas depois do *else*) é ignorado. Caso a expressão lógica resulte em *false*, o conjunto de instruções I é ignorado e o conjunto de instruções II é executado. Note que, independentemente do resultado da expressão, sempre haverá um conjunto de instruções que será executado e outro que será ignorado.

Exemplo de uso: fazer um programa que determine se uma equação do segundo grau possui ou não raízes reais.

Uma equação do segundo grau é comumente representada por $ax^2 + bx + c = 0$; possui raízes reais quando $b^2 - 4ac$ é maior ou igual a zero. Assim, uma possível solução para o problema é apresentada pela **Figura 3.9**.



```

1 import java.util.Scanner;
2 public class Equacao2Grau
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         double a,b,c; //declaração de variáveis
7         System.out.println("Digite os coeficientes da equação");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         a = ent.nextDouble(); //lendo um número e armazenando
10        b = ent.nextDouble(); //lendo um número e armazenando
11        c = ent.nextDouble(); //lendo um número e armazenando
12
13        if(b*b-4*a*c>=0) //verificando condição
14            System.out.println("A equação possui raízes reais."); // mensagem
15        else
16            System.out.println("A equação não possui raízes reais."); // mensagem
17
18    } //Fim da Função main
19 } //Fim da Classe

```

Figura 3.9: Programa para verificar se uma equação do segundo grau possui raízes reais.

O operador “?:”

O operador *?:* (também chamado de operador condicional) está intimamente ligado à estrutura *if-else*. Este operador é utilizado para montar uma expressão condicional em uma única linha e sua sintaxe é a seguinte:

(<Condição>) ? <resultado I>: <resultado II>

A execução deste operador procede da seguinte forma: primeiro, a condição é avaliada. Caso ela resulte em *true*, o resultado do operador *?:* será o resultado I. Caso a condição resulte em *false*, o resultado do operador será o resultado II.

Veja um exemplo de programa que utiliza este operador:

```

1 import java.util.Scanner;
2 public class OperadorCondicional
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int x,z; //declaração de variáveis
7         System.out.println("Digite um número");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9
10        z = ent.nextInt(); //lendo um número e armazenando
11
12        x=(z>0)?1:-1; //Usando operador condicional para atribuir um valor a x
13
14        System.out.println("x="+x); // mensagem
15    } //Fim da Função main
16 } //Fim da Classe

```

Figura 3.10: Uso do operador condicional. Observe ainda que as linhas 9 e 11 estão em branco. Isso é meramente uma questão de organização pessoal. Cada um pode deixar linhas em branco, ou não. Isso não influencia a construção do programa.

Neste exemplo, x recebe um valor dependendo do valor digitado para z. Caso z seja maior que zero, x assume o valor 1. Caso contrário, x assume -1. O trecho da linha 12 é equivalente a escrever o seguinte trecho:

```

if(z>0)

    x=1;

else

    x=-1;

```

Atividade final

Atende aos objetivos 1 e 2

- Faça um programa, baseado no programa da **Figura 3.9**, que, além de verificar se existem raízes reais, calcule quais são as raízes da equação informada, caso a equação possua raízes reais.

b) Faça um programa que verifique se um determinado número informado pelo usuário é par ou ímpar, usando o operador condicional.

c) Faça um programa que leia a idade de um indivíduo e determine, de acordo com a legislação em vigor atualmente, se este indivíduo é maior ou menor de idade.

d) O Índice de Massa Corporal (IMC) é uma medida utilizada pelos profissionais de saúde para avaliar se um indivíduo está ou não acima do peso. Para calcular este índice, dividimos a massa do indivíduo, chamada de m (obtida em kg por uma balança) pelo quadrado da sua altura, chamada de h (medida em metros), ou seja, $IMC = m/h^2$. Para os homens, caso o IMC seja maior que 26,4, o indivíduo é considerado acima do peso. Para as mulheres, esse limiar é 25,8. Faça um programa para calcular o IMC de pessoas do mesmo sexo que você. O algoritmo deve ler as informações necessárias (massa e altura), calcular o IMC e exibir uma mensagem informando se o indivíduo está ou não acima do peso ideal.

Resposta comentada

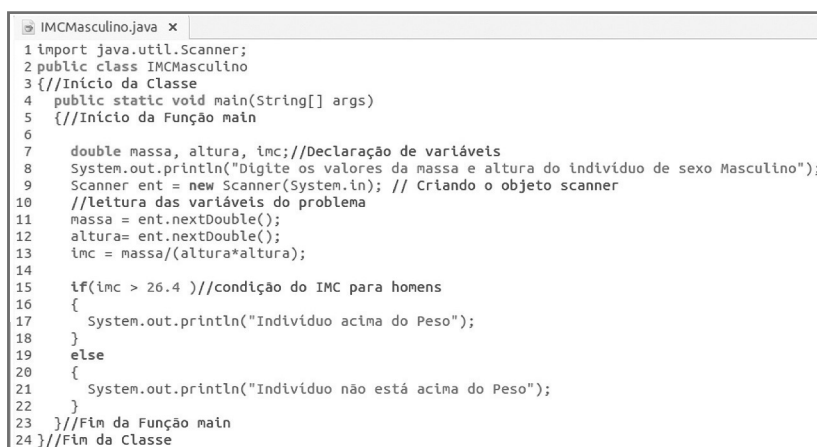
a) Para determinar as raízes de uma equação do segundo grau, devemos utilizar a fórmula de Bhaskara, que diz que a primeira raiz, chamada x_1 , é dada por $x_1 = \frac{-b + \sqrt{\Delta}}{2a}$ e a segunda raiz, denominada x_2 , é dada por $x_2 = \frac{-b - \sqrt{\Delta}}{2a}$, onde delta é dado por $\Delta = b^2 - 4ac$. Para fazer

uma raiz quadrada, basta usar o método *sqrt* da classe *Math* (Aula 2) ou fazer uma potenciação fracionária, ou seja, $\sqrt{b} = b^{1/2}$. Para fazer isso, use o método *pow*, da classe *Math*, que tem a seguinte sintaxe: *Math.pow* (base, expoente), sendo base e expoente números reais.

b) Para verificar se um número é par ou ímpar, basta verificar se ele é múltiplo de dois, utilizando o operador modulus (%).

c) A mesma estrutura utilizada no item anterior pode ser usada neste item, mudando apenas a condição e as mensagens a serem impressas.

d) O primeiro passo para resolver esta questão é perceber que os valores de massa e altura e do limiar de IMC são números reais. Assim, uma possível solução para a questão, considerando-se o sexo masculino, é dada a seguir. Uma solução para o sexo feminino é facilmente obtida trocando a mensagem inicial e substituindo o valor 26,4 para 25,8 .



```

1 import java.util.Scanner;
2 public class IMCMasculino
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6
7         double massa, altura, imc; //Declaração de variáveis
8         System.out.println("Digite os valores da massa e altura do indivíduo de sexo Masculino");
9         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
10        //leitura das variáveis do problema
11        massa = ent.nextDouble();
12        altura = ent.nextDouble();
13        imc = massa/(altura*altura);
14
15        if(imc > 26.4) //condição do IMC para homens
16        {
17            System.out.println("Indivíduo acima do Peso");
18        }
19        else
20        {
21            System.out.println("Indivíduo não está acima do Peso");
22        }
23    } //Fim da Função main
24 } //Fim da Classe
  
```

Figura 3.11: Programa para calcular o IMC de pessoas do sexo masculino.

Resumo

Nesta aula, você aprendeu a lidar com expressões lógicas simples (que retornam *verdadeiro* ou *falso*) e a utilizar estas expressões em estruturas de desvio condicional, simples e compostas. Também foi visto que, dependendo do resultado de uma expressão lógica, um determinado conjunto de instruções pode ser executado ou ignorado, dependendo da estrutura de desvio utilizada.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com expressões lógicas mais elaboradas e como construir estruturas de desvio encadeadas e o uso de aninhamento.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

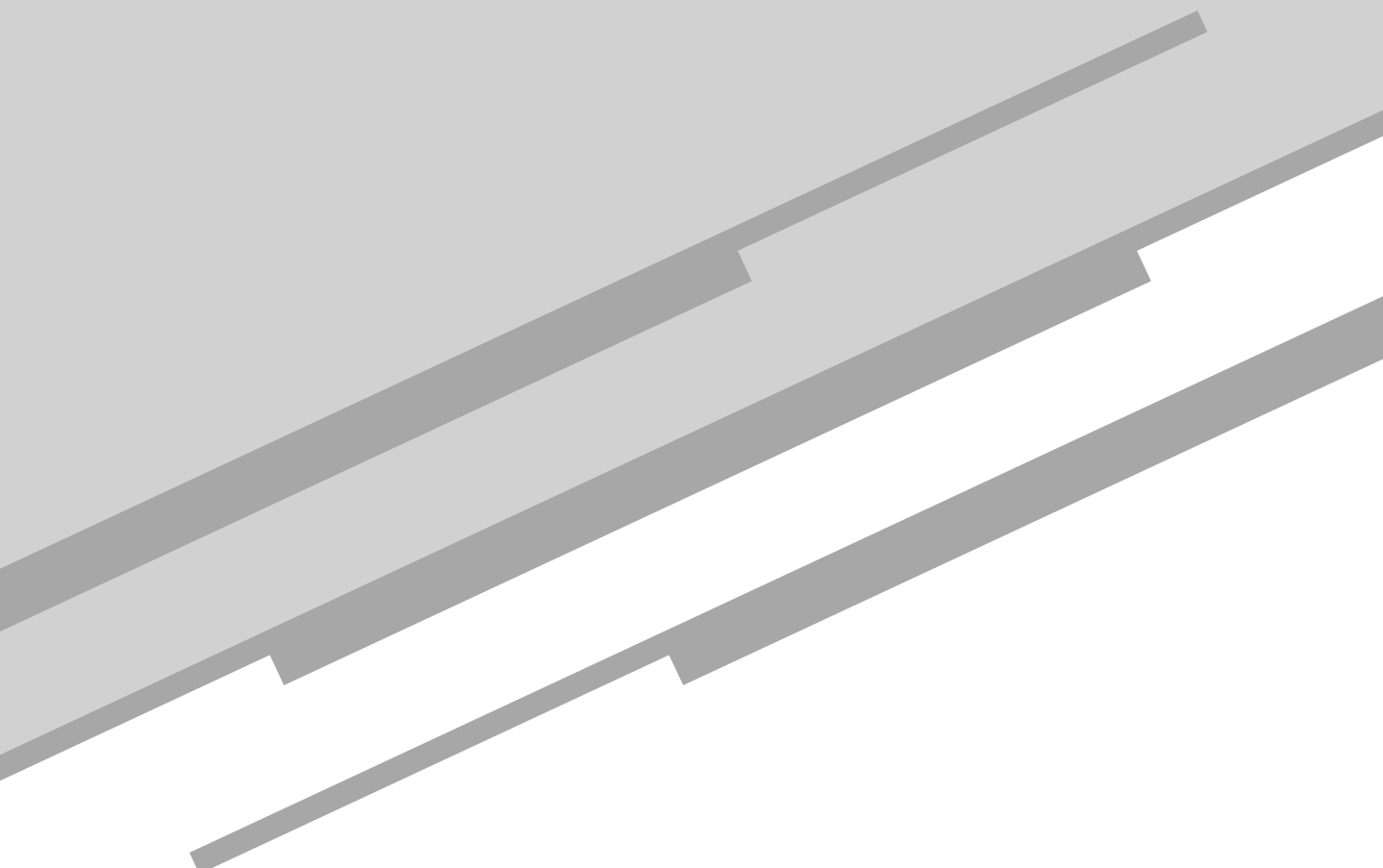
CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. et al. *Programação estruturada de computadores*. 2^a ed. Rio de Janeiro: Guanabara, 1989.

Aula 4

Desvio condicional – Parte II



Meta

Expor os conceitos avançados de desvio condicional.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. reconhecer o conceito e o uso de expressões lógicas avançadas;
2. desenvolver programas capazes de fazer desvios condicionais que utilizem expressões lógicas e aninhamento.

Expressões lógicas utilizando conjunção, disjunção e negação

Nas aulas anteriores, os programas propostos utilizavam expressões simples nas estruturas de desvio. Contudo, em muitos problemas da vida real, são necessárias análises de múltiplos valores e múltiplas variáveis para se tomar uma decisão. Para lidar com este tipo de situação, devem ser usadas operações lógicas. A linguagem Java implementa diversas destas operações:

- negação;
- conjunção;
- conjunção condicional;
- disjunção;
- disjunção condicional;
- disjunção exclusiva.

Veremos todas estas operações em detalhes. Começaremos por aquelas que existem em qualquer linguagem de programação: negação, conjunção e disjunção.

Negação

A negação é a mais simples das três operações lógicas básicas. Esta operação tem como operador a “exclamação” (!) e utiliza a seguinte sintaxe: *!operando*, e simplesmente inverte o valor lógico de um operando. Na **Tabela 4.1**, podemos observar os possíveis valores de operando e o resultado da negação do operando (*!operando*). Repare que, quando negado, o valor do operando é invertido.

Tabela 4.1: Tabela verdade para a operação de negação.

operando	!operando
true	false
false	true

Caso uma variável lógica chamada *a* deva ser negada, a expressão montada deve ter a forma *!a* e ser lida como *não a*. Neste exemplo, caso o valor de *a* seja *true*, o resultado de *!a* é *false* e vice-versa. Também é importante observar que a negação é uma **operação unária**.

Operação unária

é a que só possui um operando. Uma operação unária muito conhecida é a inversão de sinal na matemática, que é feita através do operador “-”.

Operações binárias

Nas operações binárias, são utilizados dois operandos, ou seja, são necessários dois valores lógicos para se fazer uma conjunção. Várias operações conhecidas são binárias, sendo as mais comuns: soma, subtração, multiplicação e divisão. Para realizar uma soma, por exemplo, é necessário ter dois valores para somar.

Conjunção

A conjunção, também chamada de operação *e*, tem por finalidade unir operandos de modo que uma expressão resulte em *false* quando pelo menos um dos operandos for *false*. Esta é uma **operação binária**; seu operador é o “*e* comercial” (&) e sua resolução é sempre feita da esquerda para a direita. A conjunção de dois operandos *a* e *b* pode ser feita de dois modos:

- $a \& b$ (lê-se “a e b”);
- $b \& a$ (lê-se “b e a”).

Podemos observar na **Tabela 4.2** que a conjunção só retorna verdadeiro quando os dois operandos são verdadeiros. Assim, caso um dos operandos seja falso, a conjunção sempre retorna falso.

Tabela 4.2: Tabela verdade da conjunção para dois operandos.

Operando 1	Operando 2	Operando 1 Operando 2
False	False	False
False	True	False
True	False	False
True	True	True

Disjunção

A disjunção, também conhecida como operação *ou*, tem por finalidade unir operandos de modo que, caso um deles seja *true*, toda a expressão resultará em *true*. Assim como a conjunção, esta operação é binária. Ela tem como operador a “barra vertical” (|), e sua resolução é sempre feita da esquerda para a direita. A disjunção de duas variáveis *a* e *b* pode ser feita de dois modos:

- $a | b$ (lê-se “a ou b”);
- $b | a$ (lê-se “b ou a”).

Podemos observar na **Tabela 4.3** a operação de disjunção entre dois operandos. Note que, se um dos operandos é verdadeiro, a disjunção retorna verdadeiro. Sendo assim, o único modo de uma disjunção retornar falso é quando todos os operandos são falsos.

Tabela 4.3: Tabela verdade da disjunção para dois operandos.

Operando 1	Operando 2	Operando 1 Operando 2
False	False	False
False	True	True
True	False	True
True	True	True

Embora algumas vezes utilizemos operadores lógicos e aritméticos juntos, é importante ressaltar que os tipos de dados com que cada operação trabalha são diferentes, assim como a precedência destas operações. Por exemplo, vamos avaliar a seguinte expressão:

$$3 + 4 > 6 \ \& \ 2 * 2 > 1$$

Em Java, a resolução da expressão é feita da esquerda para a direita, porém a precedência dos operadores também deve ser observada. A precedência dita como os operandos devem ser agrupados, e não exatamente a ordem de resolução das operações. Em grande parte dos casos, o agrupamento de operandos e a resolução levam a uma única sequência de resolução de operações, mas isso não ocorre em todos os casos.

Em algumas expressões podem surgir expressões menores que podem ser resolvidas independentemente da precedência, sem que isso interfira no resultado. Por exemplo, vejamos a expressão $3 * 4 + 2 - 1$. Neste caso, resolver primeiro a expressão $3 * 4$ (resultando em $12 + 2 - 1$) ou a expressão $2 - 1$ (resultando em $3 * 4 + 1$) não interfere no resultado. A precedência só dita que o número 4, ao ser utilizado em duas operações distintas (multiplicação e soma), seja agrupado com o operador da operação de maior precedência (neste caso, a multiplicação). Seguindo esta lógica, ao tomar a expressão $3 + 4 > 6 \ \& \ 2 * 2 > 1$, e agrupar seus operandos de acordo com a precedência das operações, resultaria na seguinte expressão:

$$(((3 + 4) > 6) \ \& \ ((2 * 2) > 1))$$

Operadores aritméticos têm precedência maior que operadores relacionais, que têm precedência maior que operadores de igualdade que, por sua vez, têm precedência maior que operadores lógicos. Há algumas exceções, tais como a inversão de sinal e a operação de negação, que possuem a maior precedência entre os operadores vistos até o momento.

A resolução das expressões em Java, depois que os operadores são “agrupados”, é feita da esquerda para a direita. Neste caso, a primeira

operação a ser feita será aquela delimitada pelo primeiro “)” encontrado, que é a soma de 3 e 4, e não a multiplicação de 2 por 2, como se pode pensar a princípio.



Note que este mecanismo de resolução de expressões pode variar de linguagem para linguagem.

Sendo assim:

- As expressões aritméticas são avaliadas primeiramente, o que faz com que a expressão anterior seja reduzida para: $7 > 6 \ \& \ 2 * 2 > 1$.
- Na próxima avaliação, é resolvida a expressão relacional mais à esquerda, que resulta em: *verdadeiro* & $2 * 2 > 1$.
- Neste ponto, ao aplicar a regra de precedência, encontramos uma única forma de resolução. Neste caso, a multiplicação é resolvida primeiro, ficando: *verdadeiro* & $4 > 1$.
- Na sequência, a operação relacional deve ser resolvida, o que resulta em: *verdadeiro* & *verdadeiro*.
- Por último, a operação lógica é efetuada. Como os dois operandos são verdadeiros, o resultado final da expressão é verdadeiro.



É importante mencionar que, assim como nas expressões aritméticas, os parênteses podem ser utilizados em expressões lógicas para mudar a ordem em que as operações são executadas.

Atividade 1

Atende aos objetivos 1 e 2

1. Avalie o resultado das seguintes expressões:

a) $a - 2 > 0 \ \& \ b$, sendo $a = 6$ e $b = \text{false}$

b) $b - 3! = 0 \mid \text{false}$, sendo $b=0$

c) $a > 3 \ \& \ !(3 > 4) \ \& \ \text{true}$, sendo $a = 5$

2. Faça um programa que *leia* três números e que seja capaz de *verificar* se estes números representam *ou não* os lados de um triângulo.

3. Faça um programa que receba:

- as notas de três provas;
- a quantidade de faltas de um aluno;
- o número total de aulas ministradas.

O programa deve calcular a média das notas e a frequência do aluno em porcentagem. Caso a média seja inferior a 6 ou a frequência, inferior a 75%, o programa deve imprimir que o aluno está reprovado. Caso contrário, deve imprimir que o aluno está aprovado.

Resposta comentada

1. a) Substituindo os valores das variáveis, a expressão se torna $6 - 2 > 0$ & *false*. Operadores aritméticos são resolvidos primeiro, logo temos $4 > 0$ & *false*. Dentre os operadores restantes, o de maior precedência é o relacional, logo temos *true* & *false*, que resulta em *false*.

b) Como *b* vale zero, a expressão aritmética $b - 3$ resulta em -3, que é diferente de zero; *true* | *false*, que resulta em *true*.

c) Sendo 5 o valor de *a*, e analisada as precedências dos operadores, a expressão $5 > 3$ é avaliada primeiro, resultando em *true* & $!(3 > 4)$ & *true*. Na sequência, pela presença dos parênteses, deve ser avaliada a expressão $3 > 4$, o que resulta em *true* & $!(false)$ & *true*. A negação tem maior precedência, logo a expressão fica *true* & *true* & *true*. Neste ponto, a expressão é resolvida da esquerda para a direita e resulta em *true* & *true*, que resulta em *true*.

2. Sendo *a*, *b* e *c* os lados do triângulo, devemos analisar as condições $a + b > c$, $a + c > b$ e $b + c > a$. Como a propriedade diz que a soma de quaisquer dois lados tem que ser maior que o terceiro, as três condições devem ser verdadeiras. Assim, devemos usar conjunções para unir as três condições. Uma possível solução é dada a seguir:

```

1 import java.util.Scanner;
2 public class Triangulo
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         double a,b,c; //declaração de variáveis
7         System.out.println("Digite 3 números");
8         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
9         a = ent.nextDouble(); //lendo um número e armazenando em a
10        b = ent.nextDouble(); //lendo um número e armazenando em b
11        c = ent.nextDouble(); //lendo um número e armazenando em c
12        if(a+b>c & b+c >a & a+c >b)
13            System.out.println("Os números formam um triângulo");// mensagem caso seja triângulo
14        else
15            System.out.println("Os números não formam um triângulo");//mensagem caso não seja triângulo
16
17        System.out.println("Fim do programa");// mensagem
18    } //Fim da Função main
19 } //Fim da Classe

```

Figura 4.1: As três condições devem ser verdadeiras.

3. Primeiramente devemos calcular a média das notas e a porcentagem de frequência, usando a regra de três. Feito isso, devemos montar uma expressão lógica contendo as regras de aprovação. Uma possível solução é mostrada a seguir:

```

NotasFrequencia.java x
1 import java.util.Scanner;
2 public class NotasFrequencia
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         double n1,n2,n3; //declaração de variáveis
7         int numeroAulas, faltas;
8         System.out.println("Digite as 3 notas");
9         Scanner ent = new Scanner(System.in);
10        n1 = ent.nextDouble();
11        n2 = ent.nextDouble();
12        n3 = ent.nextDouble();
13        System.out.println("Digite o número de aulas dadas e o número de faltas");
14        numeroAulas = ent.nextInt();
15        faltas = ent.nextInt();
16        double media = (n1+n2+n3)/3.0;
17        double frequencia = 100.0 - (100.0*faltas)/numeroAulas;
18
19        if(media < 6.0 | frequencia < 75.0) //condição para reprovação
20            System.out.println("O aluno está reprovado");
21        else
22            System.out.println("O aluno está aprovado");
23
24        System.out.println("Fim do programa");// mensagem final
25    } //Fim da Função main
26 } //Fim da Classe

```

Figura 3.2: Devemos calcular a média das notas e a porcentagem de frequência, para depois montar uma expressão lógica contendo as regras de aprovação.

O cálculo de frequência é feito a partir do número de faltas. Quando encontramos o percentual de faltas (feito a partir de uma regra de três), podemos calcular facilmente a frequência:

Percentual Presente = 100% – Percentual Ausente

Esta equação corresponde à linha 17 do programa.

Disjunção exclusiva

A disjunção exclusiva, também conhecida como operação *xor*, tem por finalidade unir dois operandos de modo que, se apenas um deles for *true*, toda expressão resulta em *true*. Esta operação é binária, tem como operador “acento circunflexo” (^), e sua resolução é sempre feita da esquerda para a direita. A disjunção exclusiva de duas variáveis *a* e *b* pode ser feita de dois modos:

- $a \wedge b$ (lê-se “a xor b”);
- $b \wedge a$ (lê-se “b xor a”).

Tabela 4.4: Tabela verdade da disjunção exclusiva para dois operandos

Operando 1	Operando 2	Operando 1 \wedge Operando 2
False	False	False
False	True	True
True	False	True
True	True	False

Note que a tabela verdade do operador \wedge é a mesma do operador $!=$. No entanto, estes operadores não podem ser comutados sem análise prévia. Isto ocorre porque a precedência destes operadores é diferente. Este assunto será abordado em maiores detalhes mais adiante.

Conjunção condicional e disjunção condicional

A conjunção condicional, que utiliza o operador `&&`, e a disjunção condicional, que utiliza o operador `||`, possuem as mesmas tabelas verdades que a Conjunção e a Disjunção, respectivamente. Isto significa que, em uma expressão lógica, o resultado de $a \& b$ e o resultado de $a \&\& b$ serão iguais. Entretanto, existem algumas diferenças no funcionamento das operações condicionais. A primeira diferença é a velocidade de execução. Suponha a expressão:

$$1 > 2 \& 3 > 2$$

Para descobrir o valor desta expressão, o computador a avalia da esquerda para a direita, considerando as precedências. Neste caso, os operadores relacionais têm precedência maior. Logo, após a análise da primeira operação, a expressão fica:

$$false \& 3 > 2$$

Após a avaliação da segunda operação, a expressão fica:

$$false \& true$$

Por último, a conjunção é resolvida, e a expressão resulta em *false*.

Agora suponha que esta mesma expressão seja escrita usando o operador de conjunção condicional, ou seja:

$1 > 2 \ \&\& \ 3 > 2$

A primeira avaliação ocorre do mesmo modo, e tem o mesmo resultado. Assim, a expressão fica:

$false \ \&\& \ 3 > 2$

A diferença ocorre neste ponto. Ao perceber que o operando é *false*, a conjunção condicional já é capaz de determinar o resultado da expressão sem precisar avaliar o restante da expressão. Logo, o resultado da expressão é *false*, pois a conjunção de *false* com qualquer outro termo sempre resulta em *false*. Este tipo de avaliação é chamado de avaliação em curto-circuito e também ocorre na disjunção condicional. Porém, na disjunção condicional, o valor crítico é *true*, pois a disjunção de *true* com qualquer outro termo sempre resulta em *true*.

A avaliação em curto-circuito promove um ganho de desempenho considerável se as expressões lógicas a serem avaliadas forem longas ou complexas. Neste sentido, este operador deve ser usado sempre que possível. Apesar do ganho de desempenho, se mal utilizado, este operador pode gerar resultados inesperados. Veja o seguinte exemplo:

Suponha que existam dois valores numéricos a , b . Você quer verificar se a/b é maior que 1, somente quando b for diferente de zero. Note, que não é possível montar uma expressão lógica coerente usando a conjunção comum. A expressão lógica $a/b > 1 \ \& \ b! = 0$ teria os dois lados avaliados independentemente do valor de b , e uma divisão por zero poderia ocorrer. Por outro lado, com o uso da disjunção condicional, é possível montar a expressão $b! = 0 \ \&\& \ a/b > 1$, que resolveria o problema.

Note que você quer verificar se b é diferente de zero primeiro, antes de avaliar o resultado de a/b . Neste caso, se b for zero, a avaliação em curto-circuito impede que a divisão por zero seja feita, pois $b! = 0$ resulta em *false* e a conjunção de *false* e qualquer outra coisa resulta em *false*. Porém, a expressão $a/b > 1 \ \&\& \ b! = 0$ também possui a falha da expressão $a/b > 1 \ \& \ b! = 0$, mesmo utilizando a avaliação em curto-circuito. Portanto, a ordem em que as expressões são posicionadas em uma expressão com curto-circuito é relevante.

Outro fator a ser considerado é que as precedências dos operadores $\&\&$ e $\|$ são menores que as dos operadores $\&$ e $|$, o que pode confundir na hora de montar expressões. Note que, para uma expressão do gênero $a \ \&\& \ b \ \& \ c$, os operadores são agrupados da seguinte maneira $(a \ \&\& \ (b \ \& \ c))$, e não $((a \ \&\& \ b) \ \& \ c)$. A diferença entre as duas expressões é a seguinte: na primeira, se a resultar em *false*, b e c não são avaliados. Já na segunda, caso a resulte em *false*, b não é avaliado, mas c sim.



Precedência dos operadores

A Tabela 4.5 apresenta os operadores estudados até o momento, seus tipos e sua precedência relativa. Operadores na mesma linha possuem a mesma precedência, e operadores com precedência maior são executados primeiro. Note que o operador de atribuição é o que tem a menor precedência, logo todas as expressões são avaliadas antes de se fazer uma atribuição.

Tabela 4.5: Precedência dos operadores vistos até o momento.

Operador	Tipo	Precedência
-, !	Unário	Mais alta
*, /, %	Binário	-
+, -	Binário	-
>, >=, <, <=	Binário	-
==, !=	Binário	-
&	Binário	-
^	Binário	-
	Binário	-
&&	Binário	-
	Binário	-
?:	Ternário	-
=	Binário	Mais Baixa

A estrutura *if-else-if*

Quando temos situações em que múltiplas avaliações são necessárias, podemos utilizar a estrutura *if-else-if*. A **Figura 4.3** apresenta a sintaxe desta estrutura:


```

if(<condição 1>)
{
  <conjunto de instruções 1>
}
else if (<condição 2>)
{
  <conjunto de instruções 2>
}
...
else if(<condição n>)
{
  <conjunto de instruções n>
}
else
{
  <último conjunto de instruções>
}

```

Figura 4.3: Sintaxe da estrutura *if-else-if*.

Nesta estrutura, as condições são avaliadas de cima para baixo, e no máximo um dos conjuntos de instruções pode ser utilizado. Caso a primeira condição resulte em *true*, o *Conjunto Instruções 1* será executado e os demais conjuntos ignorados. Caso ela resulte em *false*, a segunda condição será avaliada. Resultando em *true*, o *Conjunto de Instruções 2* é executado e o restante ignorado. Caso esta também resulte em *false*, a terceira condição será avaliada, e assim sucessivamente. O *else* final é opcional e é executado somente quando todas as condições resultarem em *false*. Caso não exista um *else* final e todas as condições resultarem em *false*, então nenhum dos conjuntos de instruções será executado.

Atividade 2

Atende ao objetivo 1

Faça um programa que utilize a seguinte tabela de conversão de notas em conceitos. O programa deve ler a nota de um aluno e dizer qual o conceito obtido.

Condição	Conceito
Nota ≥ 90	A
$75 \leq \text{Nota} < 90$	B
$60 \leq \text{Nota} < 75$	C
Nota < 60	D

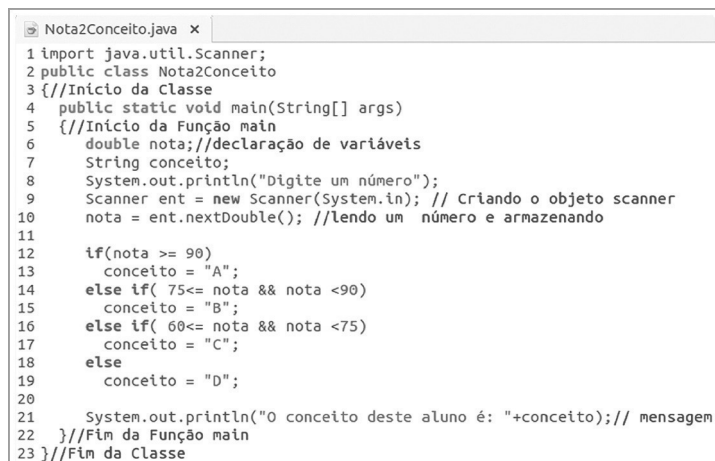
Resposta comentada

As condições deste problema são simples de se resolver, porém há um pequeno detalhe que deve ser discutido:

A condição $75 \leq \text{Nota} < 90$ é perfeitamente compreensível para seres humanos. A notação $75 \leq \text{Nota} < 90$ é uma forma compacta de se dizer que a nota deve ser maior ou igual a 75 e menor que 90, simultaneamente. Porém, esta não é uma expressão válida em Java (assim como em muitas outras linguagens). Se tentarmos resolver a expressão, encontraremos inconsistências.

Por exemplo: Imagine nota com valor 80. A expressão se torna $75 \leq 80 < 90$. Da esquerda para a direita, teremos $\text{true} < 90$, que é uma operação semanticamente incorreta, pois o operador relacional $<$ espera dois valores comparáveis entre si (o que não ocorre neste caso).

Dadas estas considerações, uma possível solução para este problema é a seguinte:



```

1 import java.util.Scanner;
2 public class Nota2Conceito
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         double nota; //declaração de variáveis
7         String conceito;
8         System.out.println("Digite um número");
9         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
10        nota = ent.nextDouble(); //lendo um número e armazenando
11
12        if(nota >= 90)
13            conceito = "A";
14        else if( 75<= nota && nota <90)
15            conceito = "B";
16        else if( 60<= nota && nota <75)
17            conceito = "C";
18        else
19            conceito = "D";
20
21        System.out.println("O conceito deste aluno é: "+conceito); // mensagem
22    } //Fin da Função main
23 } //Fin da Classe
  
```

Figura 4.5: Solução com uso da estrutura *if-else-if*.

A estrutura *switch-case*

A última estrutura condicional que veremos é a estrutura *switch-case*, que avalia o resultado de uma expressão e enumera os resultados de interesse. Cada resultado de interesse deve ser posto em um *case*. Isto faz com que, caso o resultado da expressão seja o valor expresso no *case*, todos os comandos abaixo deste *case* sejam executados. Caso nenhum dos *cases* enumerados seja o resultado da expressão, então o conjunto de comandos em *default* será executado (assim como o último *else* da estrutura *if-else-if*, o *default* é opcional na estrutura *switch-case*). A **Figura 4.6** mostra a sintaxe desta estrutura.

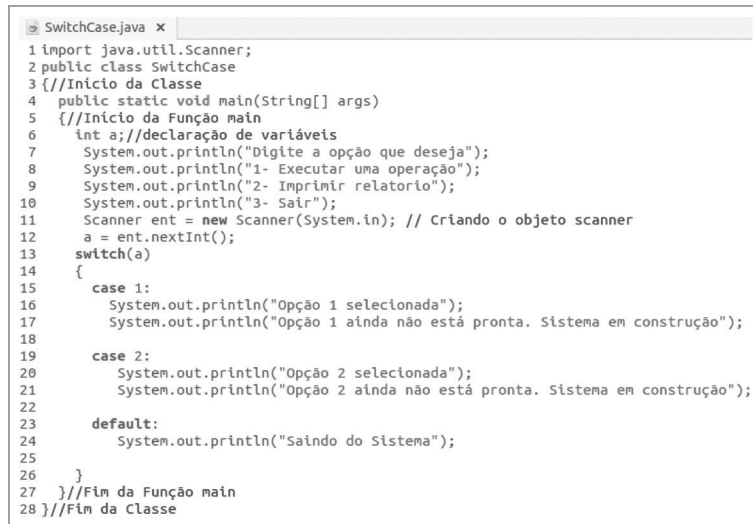
```
switch(<expressão>)  
{  
  case r1:  
    <conjunto de comandos para executar se expressão resultar em r1>  
  
  case r2:  
    <conjunto de comandos para executar se expressão resultar em r2>  
  
  ...  
  case rn:  
    <conjunto de comandos para executar se expressão resultar em rn>  
  
  default:  
    <conjunto de comandos para executar se expressão resultar em um valor não enumerado>  
}
```

Figura 4.6: Sintaxe da estrutura *switch-case*.

Note que esta estrutura é menos geral que a estrutura *if-else-if*, pois ela trabalha com a enumeração dos possíveis valores de uma expressão. Logo, o resultado da expressão não pode ser um número real. Além disso, esta estrutura pode gerar execuções bem complexas.

A **Figura 4.7** mostra um dos usos mais comuns da estrutura *switch-case*, que é a avaliação de opções de menus. Note que o sistema da figura está em construção, e, por isso, ainda não há operações implantadas. Neste programa, a opção digitada pelo usuário será armazenada em uma variável, que será usada como expressão da estrutura *switch-case*. Como há três opções, três *cases* foram colocados: um para a primeira, um para a segunda e um para todas as outras (incluindo a terceira).

Ao rodar este programa, somente as linhas abaixo do *case default* serão executadas.



```

1 import java.util.Scanner;
2 public class SwitchCase
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int a; //declaração de variáveis
7         System.out.println("Digite a opção que deseja");
8         System.out.println("1- Executar uma operação");
9         System.out.println("2- Imprimir relatório");
10        System.out.println("3- Sair");
11        Scanner ent = new Scanner(System.in); // Criando o objeto scanner
12        a = ent.nextInt();
13        switch(a)
14        {
15            case 1:
16                System.out.println("Opção 1 selecionada");
17                System.out.println("Opção 1 ainda não está pronta. Sistema em construção");
18            case 2:
19                System.out.println("Opção 2 selecionada");
20                System.out.println("Opção 2 ainda não está pronta. Sistema em construção");
21            default:
22                System.out.println("Saíndo do Sistema");
23        }
24    } //Fim da Função main
25 } //Fim da Classe

```

Figura 4.7: Uso da estrutura *switch-case*.

Em grande parte das situações, a execução de instruções de múltiplos *cases* é inconveniente ou mesmo errada. Imagine uma situação onde as opções sejam referentes ao menu de um caixa eletrônico de banco. Na situação em que não se consiga isolar os *cases*, seria impossível o cliente fazer uma única transação. Imagine a inconveniência de ser obrigado a fazer uma transferência para poder tirar um extrato. Para resolver este problema, é necessário o uso de um comando específico, que interrompa a estrutura. Para este fim, Java disponibiliza o comando *break*.

Ao passar por um comando *break*, o fluxo do programa é automaticamente enviado para o fim da estrutura, fazendo com que nenhuma outra instrução da estrutura seja executada. Neste sentido, para impedir que o fluxo siga de um *case* para outro, devemos colocar um comando *break* ao final de cada *case*. A **Figura 4.8** mostra como fazer o uso do comando *break* no exemplo anterior.

```

1 import java.util.Scanner;
2 public class SwitchCase
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int a; //declaração de variáveis
7         System.out.println("Digite a opção que deseja");
8         System.out.println("1- Executar uma operação");
9         System.out.println("2- Imprimir relatório");
10        System.out.println("3- Sair");
11        Scanner ent = new Scanner(System.in); // Criando o objeto scanner
12        a = ent.nextInt();
13        switch(a)
14        {
15            case 1:
16                System.out.println("Opção 1 selecionada");
17                System.out.println("Opção 1 ainda não está pronta. Sistema em construção");
18                break;
19            case 2:
20                System.out.println("Opção 2 selecionada");
21                System.out.println("Opção 2 ainda não está pronta. Sistema em construção");
22                break;
23            default:
24                System.out.println("Saindo do Sistema");
25                break;
26        }
27    } //Fim da Função main
28 } //Fim da Classe

```

Figura 4.8: Uso da estrutura *switch-case* com o comando *break*.

Note que, ao executar este novo programa, as instruções dos *cases* 2 e *default* não são executadas, caso você selecione a opção 1. Outra observação que deve ser feita é que as posições dos *cases* não precisam estar em nenhuma ordem particular (nem mesmo o *default*). Se você posicionar os *cases* da figura anterior em qualquer ordem, notará que eles permanecem funcionando do mesmo modo.

Atividade 3

Atende ao objetivo 1

Faça um programa que leia dois números reais e um número inteiro. Sendo que:

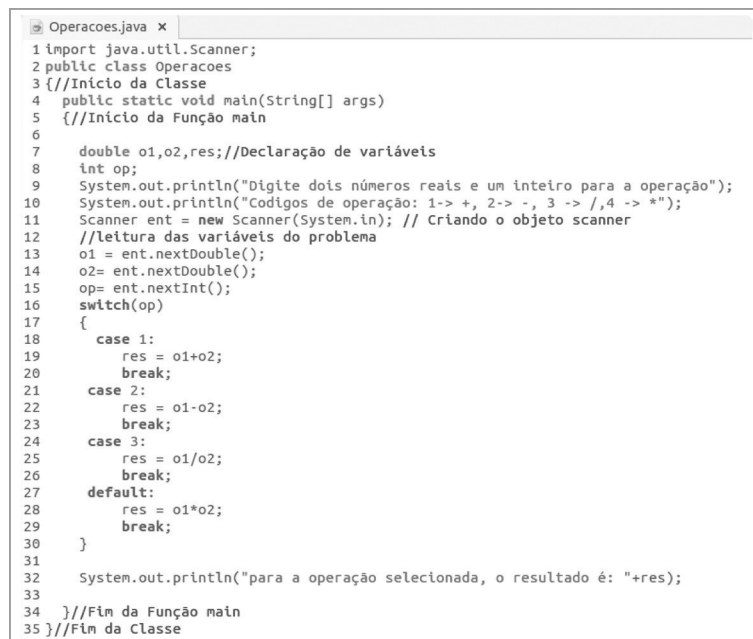
- o número inteiro é uma opção que representa uma operação a ser feita;
- os números reais são os operandos da operação em questão.

Deve-se definir que:

- caso a opção seja 1, o programa deverá calcular e imprimir a soma dos dois números reais;
- caso 2, a subtração;
- caso 3, a divisão, sendo o segundo número usado como divisor;
- a quarta opção deverá ser a multiplicação;

Resposta comentada

Podemos utilizar a estrutura *switch-case* para compor uma possível solução (imagem a seguir). Mas também é possível resolver este problema utilizando a estrutura *if-else-if*.




```

1 import java.util.Scanner;
2 public class Operacoes
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6
7         double o1,o2,res; //Declaração de variáveis
8         int op;
9         System.out.println("Digite dois números reais e um inteiro para a operação");
10        System.out.println("Codigos de operação: 1-> +, 2-> -, 3 -> /, 4 -> *");
11        Scanner ent = new Scanner(System.in); // Criando o objeto scanner
12        //leitura das variáveis do problema
13        o1 = ent.nextDouble();
14        o2= ent.nextDouble();
15        op= ent.nextInt();
16        switch(op)
17        {
18            case 1:
19                res = o1+o2;
20                break;
21            case 2:
22                res = o1-o2;
23                break;
24            case 3:
25                res = o1/o2;
26                break;
27            default:
28                res = o1*o2;
29                break;
30        }
31        System.out.println("para a operação selecionada, o resultado é: "+res);
32    } //Fim da Função main
33 } //Fim da Classe
  
```

Figura 4.9: Possível solução utilizando a estrutura *switch-case*.

Aninhamento

Em muitos casos, é necessário analisar múltiplas condições de maneira encadeada. Por exemplo, para calcular as raízes de uma equação de segundo grau, utilizando as fórmulas de Bhaskara. Primeiramente, devemos verificar se o coeficiente do termo quadrático é diferente de zero. Em seguida, temos que verificar se existem raízes reais, e só então devemos calcular as raízes. Para evitar cálculos desnecessários, podemos utilizar uma estrutura de *ifs*, assim como na **Figura 4.10**.



```

1 import java.util.Scanner;
2 public class BhaskaraAninhado
3 { //Inicio da Classe
4     public static void main(String[] args)
5     { //Inicio da Função main
6
7         double a, b, c; //Declaração de variáveis
8         System.out.println("Digite os valores dos coeficientes a,b e c respectivamente");
9         Scanner ent = new Scanner(System.in); // Criando o objeto scanner
10        //Leitura das variáveis do problema
11        a = ent.nextDouble();
12        b = ent.nextDouble();
13        c = ent.nextDouble();
14        if(a!=0)
15        {
16            double delta = b*b-4*a*c;
17            if(delta >=0 )
18            {
19                //Calculando a resposta
20                double resultado1 = (-b + Math.sqrt(delta))/(2*a) ;
21                double resultado2 = (-b - Math.sqrt(delta))/(2*a) ;
22                System.out.println("raizes:"+resultado1+ " e "+resultado2);
23            }
24            else
25            {
26                System.out.println("Não há raízes reais"); // Impressão do resultado
27            }
28        }
29        else
30        {
31            System.out.println("O termo quadrático possui coeficiente zero");
32        }
33    }
34 } //Fim da Função main
35 //Fim da Classe
  
```

Figura 4.10: Estruturas *if* aninhadas para as fórmulas de Bhaskara.

Este tipo de construção, que usa uma estrutura dentro de outra, é chamado de aninhamento e pode ser usado com de diversos tipos de estrutura, como estruturas de repetição, que serão vistas em aulas posteriores. Para compor o aninhamento corretamente, é necessário colocar uma estrutura dentro da outra, o que significa que, tendo duas estruturas condicionais, uma interna e uma externa, é necessário que todos os elementos da estrutura interna estejam dentro da estrutura externa. A ideia é similar à de construir expressões com vários parênteses. Por exemplo, imagine a seguinte expressão:

$$4*(2/(1-3))$$

Note que há dois pares de parênteses, e que a parte da equação delimitada pelos parênteses maiores está dentro da parte delimitada pelos parênteses menores. Ou seja, os parênteses maiores estão dentro dos menores.

Na **Figura 4.10**, ainda podemos observar um programa que possui duas estruturas *if-else*, uma mais interna (linhas 17 a 27) e uma mais externa (linhas 14 a 32). Note que todas as instruções da estrutura mais interna estão dentro do *if* (linhas 14 a 28) da estrutura mais externa. Isto significa que o *if-else* mais interno (linhas 17 a 27) está completamente contido dentro do *if* da estrutura mais externa. Entendendo a estrutura, você pode perceber que o fim de bloco (`}`) da linha 27 pertence à estrutura *if-else* que começa na linha 17. Consequentemente, o fim da estrutura mais externa está na linha 32.



Indentação e programas

A indentação é simplesmente o controle do recuo de onde começa cada linha de código. Podemos observar este recuo na **Figura 4.10**. Note que, como a estrutura *if-else* mais interna está dentro do *if* da mais externa, ela tem um recuo maior (mais distante da margem esquerda).

Em Java, o aninhamento é tratado simplesmente por meio do posicionamento correto das estruturas. Entretanto, a indentação é considerada uma boa prática de programação, evidente no aninhamento de estruturas.

Apesar de ser apenas uma boa prática em Java, a indentação é obrigatória em algumas linguagens, como Python por exemplo. Além disso, um código indentado é muito mais legível e de fácil entendimento. Por isso, indente o seu código e você verá que ele ficará muito mais organizado e de fácil compreensão.

Atividade 4

Atende aos objetivos 1 e 2

a) Desenvolva um programa que, dados três lados de um triângulo, classifique este triângulo como equilátero, escaleno ou isósceles e imprima o tipo correspondente.

b) Faça um programa que leia três informações de um indivíduo: Sexo, massa e altura. O programa deve calcular o índice de massa corporal (IMC), verificar e imprimir se o indivíduo está abaixo do peso, no peso ou acima do peso.

Saiba que:

- Para calcular o IMC, dividimos a massa do indivíduo (m) em kg pelo quadrado da sua altura (h) em metros, ou seja: $IMC = m/(h.h)$.
- Para os homens, caso o IMC esteja entre 20,7 e 26,4, o indivíduo está dentro do peso.
- Para as mulheres, esse intervalo correspondente é 19,1 a 25,8.
- Caso o IMC esteja na faixa abaixo destes intervalos, o indivíduo estará abaixo do peso; se estiver acima deste intervalo, ele está acima do peso.

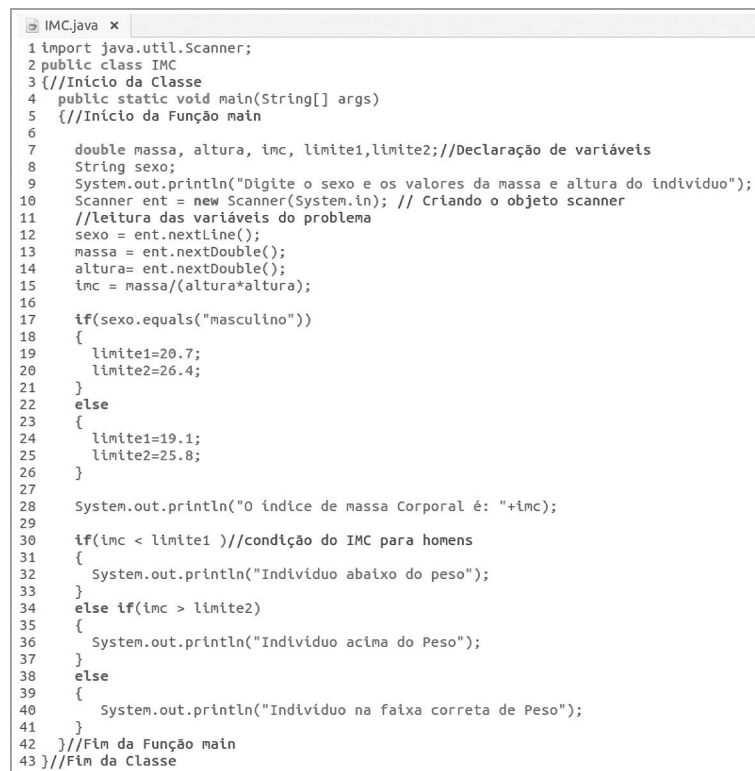
c) Crie um programa que receba a idade de uma pessoa e diga se o voto é facultativo, obrigatório ou se ela não pode votar, de acordo com a legislação eleitoral brasileira.

Saiba que:

- Pela legislação atual, um indivíduo é obrigado a votar se tiver entre 18 e 70 anos.
- O voto facultativo caso ele tenha mais de 70 e de 16 a 18 anos.
- Com menos de 16 anos, um indivíduo não pode votar.

Resposta comentada

- a) Para verificar se um triângulo é equilátero, basta verificar se todos os lados são iguais. Para verificar se ele é escaleno, todos os lados devem ser diferentes. Para verificar se o triângulo é isósceles, basta verificar se ele possui dois lados iguais. Uma estrutura de *ifs* aninhados resolve bem este problema. No *if* mais externo, verifique se o triângulo é equilátero. No *else* deste *if*, coloque outro *if-else* aninhado para verificar se ele é escaleno. Ao eliminar estas duas possibilidades a que sobra é o caso do triângulo isósceles.
- b) Uma solução possível seria, ao invés de avaliar todos os quesitos em uma mesma instrução de desvio, fazer uma divisão das avaliações em duas instruções de desvio. A primeira determina quais os limiares do IMC que serão utilizados em função do sexo do indivíduo. A segunda faz a avaliação do IMC com os limiares definidos na primeira.

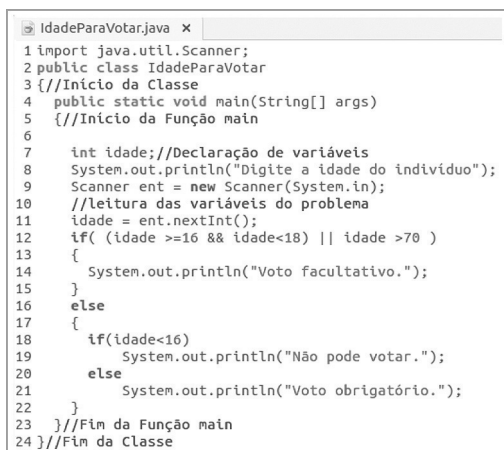


```

1 import java.util.Scanner;
2 public class IMC
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6
7         double massa, altura, imc, limite1, limite2; //Declaração de variáveis
8         String sexo;
9         System.out.println("Digite o sexo e os valores da massa e altura do indivíduo");
10        Scanner ent = new Scanner(System.in); // Criando o objeto scanner
11        //leitura das variáveis do problema
12        sexo = ent.nextLine();
13        massa = ent.nextDouble();
14        altura = ent.nextDouble();
15        imc = massa/(altura*altura);
16
17        if(sexo.equals("masculino"))
18        {
19            limite1=20.7;
20            limite2=26.4;
21        }
22        else
23        {
24            limite1=19.1;
25            limite2=25.8;
26        }
27
28        System.out.println("O índice de massa Corporal é: "+imc);
29
30        if(imc < limite1) //condição do IMC para homens
31        {
32            System.out.println("Indivíduo abaixo do peso");
33        }
34        else if(imc > limite2)
35        {
36            System.out.println("Indivíduo acima do Peso");
37        }
38        else
39        {
40            System.out.println("Indivíduo na faixa correta de Peso");
41        }
42    } //Fim da Função main
43 } //Fim da Classe
  
```

Figura 4.11: Divisão das avaliações em duas instruções de desvio.

c) Há várias maneiras de montar as condições para resolver esta questão. Uma possível solução é mostrada a seguir:



```

1 import java.util.Scanner;
2 public class IdadeParaVotar
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6
7         int idade; //Declaração de variáveis
8         System.out.println("Digite a idade do individuo");
9         Scanner ent = new Scanner(System.in);
10        //leitura das variáveis do problema
11        idade = ent.nextInt();
12        if( (idade >=16 && idade<18) || idade >70 )
13        {
14            System.out.println("Voto facultativo.");
15        }
16        else
17        {
18            if(idade<16)
19                System.out.println("Não pode votar.");
20            else
21                System.out.println("Voto obrigatório.");
22        }
23    } //Fim da Função main
24 } //Fim da Classe
  
```

Figura 4.12: Possível solução com estrutura de *ifs* aninhados.

Resumo

Nesta aula, você aprendeu a montar e interpretar expressões lógicas utilizando conjunção, disjunção, negação, disjunção exclusiva, conjunção e disjunção condicionais. Viu que estas operações podem ser utilizadas para expressar condições referentes a múltiplos valores e a múltiplas variáveis. Também foram apresentadas duas estruturas de desvio (*switch-case* e *if-else-if*) e foi mostrado como fazer o aninhamento de estruturas condicionais e que este aninhamento é utilizado quando devemos analisar uma condição dependendo do resultado de outra condição.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com situações em que devemos repetir tarefas muitas vezes sem que isso implique replicação de trechos de código.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

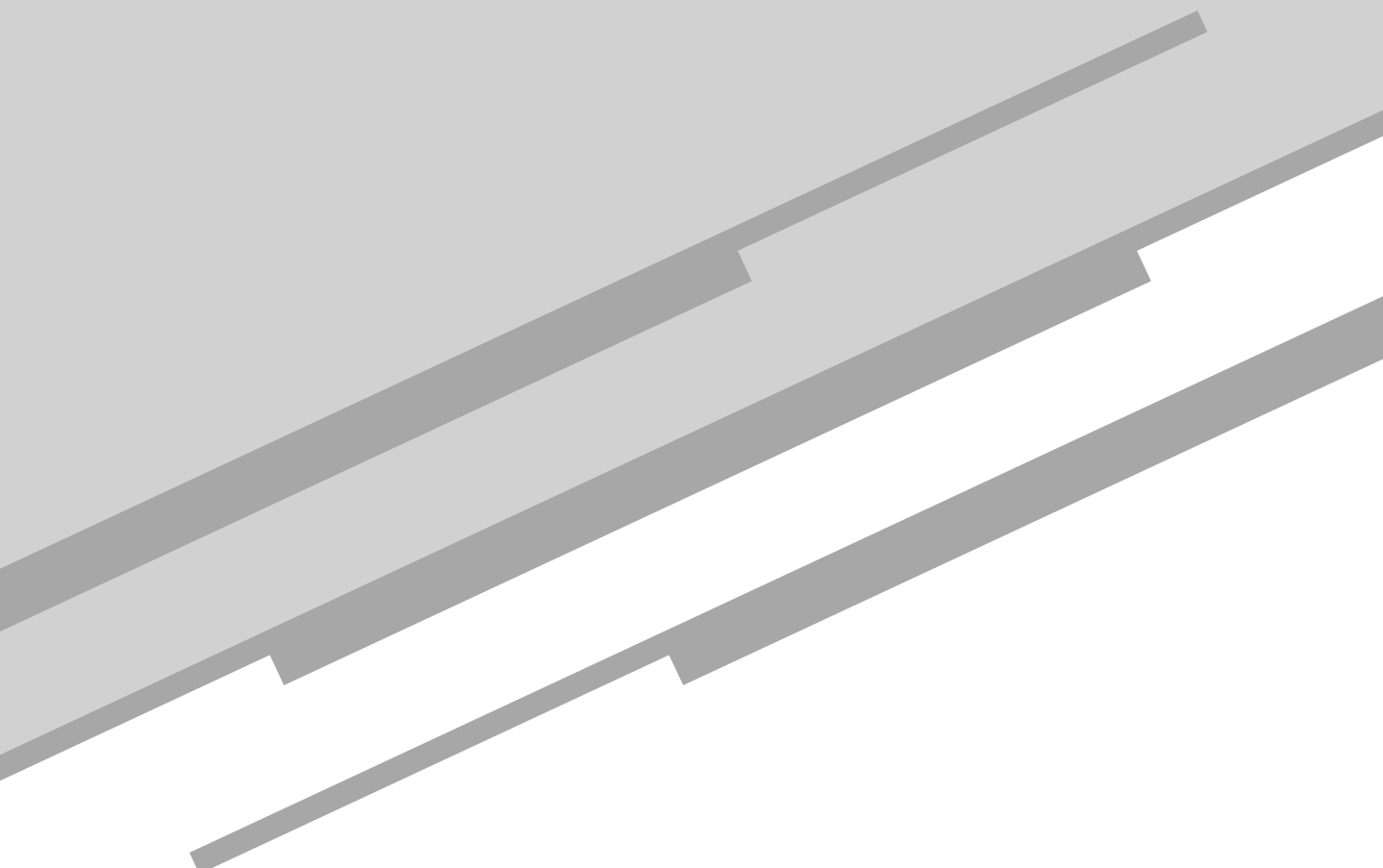
CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. et al. *Programação estruturada de computadores*. 2^a ed. Rio de Janeiro: Guanabara, 1989.

Aula 5

Repetição – Parte I



Meta

Apresentar os conceitos de repetição e uma das estruturas usadas para este fim.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. aplicar o conceito de repetição;
2. desenvolver estruturas de repetição com contador;
3. criar programas que combinam estruturas de repetição e de desvio para resolver problemas.

Repetição sem repetir

Em nossas vidas, mesmo que inconscientemente, repetimos várias vezes uma mesma tarefa. Por exemplo, no ato de comer um sanduíche há diversas ações repetitivas, que fazemos automaticamente. Primeiro, mordemos um pedaço do sanduíche. Em seguida, mastigamos a porção que foi mordida. Por último, engolimos o alimento mastigado. Estas três tarefas são executadas nesta ordem até que o sanduíche tenha se esgotado.

Outro exemplo seria um professor lançando as notas dos alunos no sistema da universidade. Para cada aluno, o professor deve conferir o número de matrícula e lançar a nota. Se existirem 60 alunos, serão 60 matrículas para conferir e 60 notas para lançar.

Em programação, um modo ingênuo de fazer repetição de tarefas é replicando trechos de código. Porém, este tipo de abordagem não é muito eficiente, pois imagine que você tem uma tarefa a ser repetida e que o trecho que faz esta tarefa possui cinco linhas. Se for necessário repetir esta tarefa seis vezes, no final, você terá 30 linhas no seu programa. Isto aumenta muito o tamanho do código e dificulta a sua legibilidade. Além disso, esta abordagem mostra-se ineficaz, quando não é possível determinar o número de vezes que a tarefa precisa ser repetida. No exemplo da tarefa de comer um sanduíche, não podemos supor quantas vezes teremos que repetir as três ações. Existem fatores que podem influenciar neste número, como, por exemplo, o tamanho do sanduíche. Outro exemplo é o sistema de *login* de uma rede social, que não pode fazer nenhuma suposição sobre quantos usuários vão acessar o sistema em um dia. Assim como um sistema que faz simulações de financiamento bancário não pode supor quantos usuários farão simulação e nem quantas simulações cada usuário fará. Um *player* de músicas não pode supor quantas músicas o usuário vai querer ouvir, dentre outros exemplos.

Para tratar destes problemas relacionados à repetição de tarefas, a computação usa estruturas de repetição. Tais estruturas evitam a replicação do código, tornando-o mais legível e elegante, e são capazes de realizar as tarefas conhecendo ou não o número de repetições necessárias. Nesta aula, veremos a estrutura de repetição *for*, também chamada de estrutura de repetição com contador. Esta estrutura é utilizada quando o número de repetições é conhecido.

Estrutura de repetição com contador

A primeira estrutura de repetição que será mostrada é a estrutura *for*, que é equivalente à estrutura *PARA*, que você aprendeu em Computação I. A sintaxe da estrutura *for* está exposta na **Figura 5.1**:

```
for(<Inicialização> ; <Critério de repetição>; <Passo> )  
{  
    <Instruções a serem repetidas>  
}
```

Figura 5.1: Sintaxe da estrutura *for*.

Esta estrutura começa com a palavra reservada *for* e repete a primeira instrução (sem { }) ou o primeiro bloco de instruções (com { }) que aparece posicionado logo depois do carácter “)” (parêntese à direita). Imediatamente após o *for*, há um par de parênteses, e entre eles estão:

- inicialização – que diz a partir de qual valor a contagem deve começar;
- critério de repetição – que diz até quando a contagem deve continuar;
- passo da contagem – que determina como a contagem é feita.

As etapas da repetição seguem a seguinte ordem:

1. inicialização: a primeira coisa a ser feita nesta estrutura é a inicialização, que é feita uma única vez no começo da repetição;
2. avaliação do critério de repetição. Se o resultado da análise *for true*, as instruções são executadas. Senão, o fluxo do programa segue imediatamente para a primeira instrução depois da repetição (este passo é executado em todas as repetições, sempre antes das instruções);
3. instruções a serem repetidas: nesta etapa, as instruções que se quer repetir são executadas;
4. passo: caso a avaliação do critério de repetição tenha resultado em *true*, é necessário fazer o passo, alterando as variáveis de contagem. Esta etapa é a última coisa a ser feita e é feita logo após a execução das instruções a serem repetidas.

Após o passo, é feita uma nova avaliação do critério de repetição (etapa 2). Se esta nova avaliação também resultar em *true*, as instruções serão repetidas novamente (etapa 3), assim como o passo (etapa 4). Em seguida, uma nova avaliação do critério de repetição será feita, e assim sucessivamente, até que a avaliação do critério de repetição resulte em *false*.



1. Cada execução das etapas 2, 3 e 4 da estrutura é chamada de **ITERAÇÃO**.
2. Apesar de estar dentro da estrutura de repetição, a inicialização não é repetida. O que é repetido são: a avaliação do critério de repetição, as instruções a serem repetidas e o passo.
3. Apesar de estar descrito no começo da estrutura, o passo é a última coisa a ser feita na repetição.

```
testeFor.java x
1 public class testeFor
2 {
3     public static void main(String[] a)
4     {
5
6         for(int i=0;i<10;i++)
7         {
8             System.out.println(i);
9         } //fim da repetição
10    } //fim da função main
11 } //fim da classe
```

Figura 5.2: Exemplo de uso da estrutura *for*.

A **Figura 5.2** mostra um exemplo de uso da estrutura *for*. Neste exemplo, a variável *i* é criada dentro da estrutura de repetição e inicializada com 0. Em seguida, é feita a comparação do valor de *i* com a constante 10. Como *i* no início tem valor 0, a expressão $i < 10$ retorna *true*.

O próximo passo é executar as instruções do bloco. Neste caso, só há uma instrução, que é a de imprimir o valor de *i*. Após esta impressão, é feito o passo da contagem. A expressão *i++* tem a mesma semântica da expressão $i = i + 1$. Logo, *i* recebe o seu próprio valor acrescido de uma unidade, ou seja, *i* passa a valer 1. Novamente, é feita a comparação do valor de *i* com a constante 10. Como *i* vale 1, a comparação $i < 10$ novamente retorna *true*.

Em seguida, deve-se imprimir o valor de *i*, ou seja, imprimir 1. O passo faz com que o valor de *i* seja modificado para 2 e, assim o algoritmo prossegue até que *i* assumo o valor 9. Neste momento, a comparação $i < 10$ retorna *true*. O valor 9 é impresso e o passo é feito, fazendo com que *i* assumo o valor 10. A comparação $i < 10$ falha neste momento, e o

fluxo do algoritmo é desviado para o fim do bloco (linha 9), terminando a repetição.

Como não há nenhuma instrução após a repetição, o algoritmo termina sua execução. Logo, este algoritmo imprime os números de 0 a 9, um em cada linha. Note que, se a mesma tarefa fosse feita através da replicação de código, seriam necessários 10 comandos “System.out.println”. Com o uso de uma estrutura de repetição, apenas três linhas foram necessárias.

Antes de falar mais sobre a repetição, vamos observar a linha 6 da **Figura 5.2**. Note que a inicialização e o passo possuem trechos de código que precisam de mais detalhamento. Note que, na inicialização, existe a criação de uma variável inteira chamada *i*. Como mencionado antes, é possível criar variáveis em vários lugares em Java e um destes lugares é a inicialização de uma repetição. Estas variáveis só existem dentro da estrutura de repetição, portanto, depois do bloco de repetição, estas variáveis não são acessíveis. Também é possível criar mais de uma variável, caso você precise fazer múltiplas contagens, mas todas devem ser do mesmo tipo.

Quanto ao passo, foi comentado que a expressão *i++* tem a mesma semântica que a expressão *i = i + 1*. Isto ocorre porque a linguagem Java tem vários operadores de atribuição. Estes operadores, assim como um exemplo de uso, estão descritos na **Tabela 5.1**, a seguir.

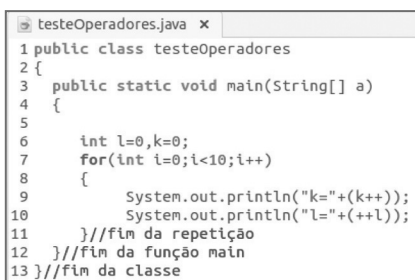
Tabela 5.1: Operadores de atribuição.

Operador	Exemplo de Uso	Expressão Equivalente
++ (pós-fixado)	<i>a++</i>	<i>a=a+1</i>
++(pré-fixado)	<i>++a</i>	<i>a=a+1</i>
--(pós-fixado)	<i>a--</i>	<i>a=a-1</i>
--(pré-fixado)	<i>--a</i>	<i>a=a-1</i>
<i>+=</i>	<i>a+=2</i>	<i>a=a+2</i>
<i>-=</i>	<i>a-=3</i>	<i>a=a-3</i>
<i>*=</i>	<i>a*=2</i>	<i>a=a*2</i>
<i>/=</i>	<i>a/=3</i>	<i>a=a/3</i>
<i>%=</i>	<i>a%=5</i>	<i>a=a%5</i>

É possível escrever uma mesma expressão usando diferentes operadores. Por exemplo, se você deseja adicionar 1 a uma variável *c* e colocar

o resultado na própria variável c , você pode fazer isso dos seguintes modos: $c = c + 1$; $c += 1$; $c++$ e $++c$. Esses operadores existem para simplificar o código, criando expressões menores.

Deve-se ressaltar que o operador $++$ e $--$ (tanto pré-fixados, quanto pós-fixados) só servem para incrementar e decrementar, respectivamente, uma variável em uma unidade. A diferença entre os operadores pré-fixados e pós-fixados está no resultado retornado pelas operações. No exemplo da **Figura 5.3** são utilizados dois, os operadores $++$, o pós-fixado e o pré-fixado, para duas variáveis distintas, k e l , respectivamente.



```

1 public class testeOperadores
2 {
3     public static void main(String[] a)
4     {
5
6         int l=0,k=0;
7         for(int i=0;i<10;i++)
8         {
9             System.out.println("k="+k++);
10            System.out.println("l="+++l);
11        } //fim da repetição
12    } //fim da função main
13 } //fim da classe

```

Figura 5.3: Uso do operador $++$ pré/pós-fixado.

Neste exemplo, uma estrutura de repetição é utilizada para repetir 10 vezes o uso de cada operador, mas, apesar de terem a mesma semântica (ambos incrementam a variável em uma unidade), o resultado das impressões das variáveis é diferente. A atividade a seguir irá lhe ajudar a compreender melhor esta diferença.

Atividade 1

Atende aos objetivos 1 e 2

Implemente o código apresentado pela **Figura 5.3** e analise as diferenças entre os valores das variáveis k e l . Você tem ideia de por que isto ocorre?

Resposta comentada

Note que, apesar de serem iniciadas com o mesmo valor, as variáveis k e l impressas apresentam divergência no valor que é impresso sempre em uma unidade. Neste caso, l é sempre uma unidade maior que k . Isto ocorre porque, com o uso do operador pós-fixado (caso da variável k), o incremento ocorre, mas o valor retornado é o antigo valor do operando. Ou seja, aumenta o valor de k em uma unidade e retorna $k - 1$. Já com o operador pré-fixado (caso da variável l), o incremento é feito e o valor retornado é o novo valor do operando. Ou seja, aumenta l em uma unidade e retorna o valor de l . Portanto, deve-se verificar cuidadosamente qual o operador adequado para cada situação.



Precedência de operadores

A tabela abaixo mostra todos os operadores vistos até o momento na ordem de sua precedência. Os operadores mais acima na tabela têm precedência maior do que os que estão abaixo. Operadores na mesma linha têm a mesma precedência.

Tabela 5.2: Operadores em ordem de precedência.

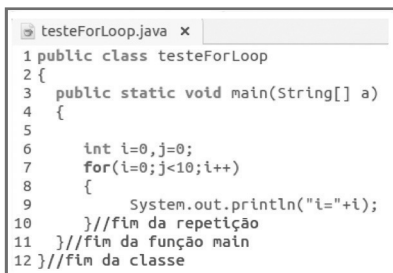
Operador	Tipo	Precedência
++ -- (pós-fixado)	Unário	Mais alta
-, !, ++ -- (pré-fixado)	Unário	-
*, /, %	Binário	-
+, -		
>, >=, <, <=	Binário	-
==, !=	Binário	-
&	Binário	-
^	Binário	-
	Binário	-
&&	Binário	-

	Binário	-
?:	Ternário	-
=	Binário	Mais Baixa

É importante mencionar que a combinação entre inicialização, critério de repetição e passo é que determina quantas vezes as tarefas serão repetidas. Por exemplo, suponha que uma determinada tarefa deva ser realizada 10 vezes. O modo mais simples de se fazer isso é contar de 1 em 1 (passo 1), começando em 1 (inicialização 1) e indo até 10 (Critério de repetição ≤ 10).

Podemos realizar esta mesma tarefa contando de inúmeras formas. Podemos começar em 10 (inicialização 10), contar até 1 (Critério de repetição ≥ 1), contando de -1 em -1 (passo -1). Podemos contar de 2 até 20 contando de 2 em 2, de 10 até 100 contando de 10 em 10 e assim sucessivamente.

Seguindo esta lógica, podemos dizer que o que determina a quantidade de vezes que uma tarefa é repetida não é a inicialização, nem o critério de repetição e nem o passo, mas sim o contexto formado por estes três argumentos.



```

1 public class testeForLoop
2 {
3     public static void main(String[] a)
4     {
5
6         int i=0,j=0;
7         for(i=0;j<10;i++)
8         {
9             System.out.println("i="+i);
10        } //fim da repetição
11    } //fim da função main
12 } //fim da classe

```

Figura 5.4: Algoritmo com repetição em *loop*.

A **Figura 5.4** mostra um mau exemplo do uso de uma estrutura de repetição. Note que a inicialização é feita em *i*, assim como o passo. No entanto, o critério de repetição é feito em *j*. Como o valor de *j* não é alterado, independentemente do número de repetições feitas, este programa irá imprimir o valor de *i* indefinidamente. Logo, a estrutura

de repetição nunca termina, o que, no jargão técnico da informática, é chamado de *loop* infinito ou, em alguns casos, simplesmente *loop*. O uso de estruturas de repetição que executam indefinidamente acaba por gerar algoritmos (e por consequência programas) mal formados, uma vez que, por definição, um algoritmo deve ser um conjunto *finito* de passos para resolver um problema.

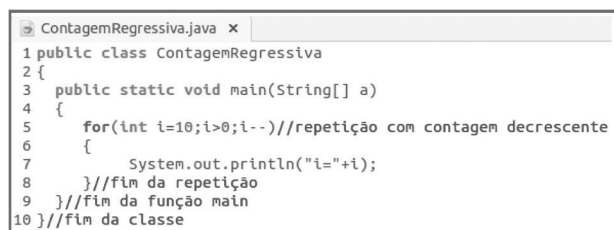
===== **Atividade 2** =====

Atende ao objetivo 3

- a) Faça um programa que imprima os números de 10 a 1 em ordem decrescente.
- b) Faça um programa que calcule a média aritmética de 10 números digitados pelo usuário.
- c) Faça um programa para calcular x elevado a n , onde x é um número real e n , um número inteiro. Ambos os números devem ser informados pelo usuário.
- d) Faça um programa que calcule o fatorial de um número qualquer digitado pelo usuário.

Resposta comentada

a) O modo mais simples de resolver esta questão é fazer um algoritmo similar ao da **Figura 5.2**, alterando o controle da repetição, para que ele faça a contagem regressivamente. A solução é mostrada a seguir:



```

1 public class ContagemRegressiva
2 {
3     public static void main(String[] a)
4     {
5         for(int i=10;i>0;i--)//repetição com contagem decrescente
6         {
7             System.out.println("i="+i);
8         }//fim da repetição
9     }//fim da função main
10 }//fim da classe

```

Figura 5.5: Programa que imprime os números de 10 a 1 em ordem decrescente.

b) Sendo X o conjunto dos números digitados, a média aritmética deste conjunto é dada pela seguinte equação:

$$\frac{1}{10} \cdot \sum_{i=1}^{10} x_i, \forall x_i \in X$$

Ou seja, devemos somar todos os elementos de X e, em seguida, dividir a soma pelo número de elementos (neste caso, 10). Como visto na disciplina de Computação I, para calcular uma soma de vários números, pode-se utilizar a propriedade de que a soma total pode ser feita por partes. Por exemplo:

Suponha a soma de quatro números: $2 + 4 + 5 + 9$. Esta soma pode ser resolvida da seguinte maneira: $(2 + 4) + 5 + 9$, que, por sua vez, é igual a $((2 + 4) + 5) + 9$. Ou seja, para resolver a soma do conjunto, basta resolver uma soma de cada vez, usando como operandos um elemento e o resultado da soma dos elementos anteriores.

Lembre-se de que a operação de soma precisa de dois operandos e que podemos usar o elemento neutro da soma como primeiro resultado. Ou seja: $2 + 4 + 5 + 9 = 0 + 2 + 4 + 5 + 9$. Isto possibilita que a soma possa ser calculada à medida que os números são inseridos.

Um programa que usa a estratégia de solução proposta é mostrado a seguir:

```

MediaAritmetica10Numeros.java x
1 import java.util.Scanner;
2 public class MediaAritmetica10Numeros
3 {
4     public static void main(String[] a)
5     {
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner
7         System.out.println("Este programa calcula a média de 10 números");//Interagindo com o usuário
8         double x, soma=0;
9         for(int i=1;i<=10;i++)//repetição com contagem decrescente
10        {
11            System.out.println("Digite o "+i+"º número");//Interagindo com o usuário
12            x=ent.nextDouble();//lendo um numero
13            soma += x;//calculando as somas parciais
14        }//fim da repetição
15        double media = soma/10.0;
16        System.out.println("A média é: "+media);//resultado
17    }//fim da função main
18 }//fim da classe

```

Figura 5.6: Cálculo da média aritmética de 10 números digitados.

Neste programa, a variável soma é utilizada para acumular os valores das somas das parcelas. Note que esta variável é inicializada com zero para que a soma possa ser feita de maneira bem-sucedida. A variável x é utilizada para ler o número que vai ser somado, e a variável i utilizada para contar quantos números foram lidos. Por último, é feita a divisão e a impressão do resultado.

c) A estratégia é basicamente a mesma do item anterior, tomando os devidos cuidados para que o elemento neutro utilizado seja o da multiplicação. Uma possível resposta é dada a seguir.

```

Potencia.java x
1 import java.util.Scanner;
2 public class Potencia
3 {
4     public static void main(String[] a)
5     {
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner
7         System.out.println("Este programa calcula x elevado a n");//Interagindo com o usuário
8         System.out.println("Digite a base e o expoente");//Interagindo com o usuário
9         double x, result=1;
10        int n;
11        x=ent.nextDouble();
12        n=ent.nextInt();
13        for(int i=1; i <= n; i++)//repetição com contagem crescente
14            result *= x;//calculando as multiplicações
15
16        System.out.println("o resultado é: "+ result);//resultado
17    }//fim da função main
18 }//fim da classe

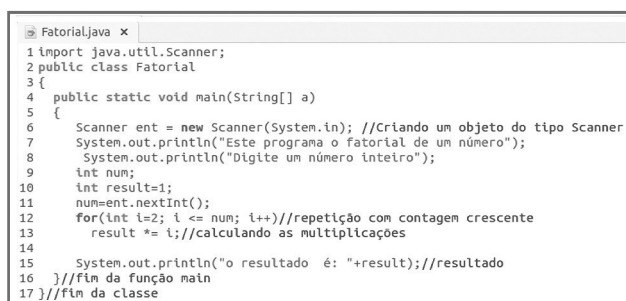
```

Figura 5.7: Programa para calcular x elevado a n , onde x é um número real e n , um número inteiro.

Note que a variável acumuladora é *result* e tem o valor inicial configurado em 1 (que é o elemento neutro da multiplicação). Para calcular a potência, devemos fazer uma série de multiplicações de um mesmo número. A potência 5^3 , por exemplo, pode ser descrita como $5 * 5 * 5$. Note que a parcela x da multiplicação é sempre a mesma. Isto ocorre porque x representa a base da potenciação. O resultado das sucessivas multiplicações é armazenado na variável *result*. Na primeira iteração, *result* vale 1 e seu valor será modificado para o resultado da multiplicação

de $result * x = 1 * 5 = 5$. Na segunda iteração, $result$ já vale 5 e seu valor será alterado para $result * x = 5 * 5 = 25$. Na terceira iteração, $result$ vale 25 e seu valor será alterado para $result * x = 25 * 5 = 125$.

d) Para calcular o fatorial de um número num , devemos fazer as multiplicações $num * (num - 1) * \dots * 2 * 1 = 1 * 2 * \dots * num$. Portanto, tanto faz multiplicar na ordem normal ou na ordem inversa. A solução apresentada calcula o resultado da multiplicação na ordem normal.



```

1 import java.util.Scanner;
2 public class Fatorial
3 {
4     public static void main(String[] a)
5     {
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner
7         System.out.println("Este programa o fatorial de um número");
8         System.out.println("Digite um número inteiro");
9         int num;
10        int result=1;
11        num=ent.nextInt();
12        for(int i=2; i <= num; i++)//repetição com contagem crescente
13            result *= i;//calculando as multiplicações
14
15        System.out.println("o resultado é: "+result);//resultado
16    }//fim da função main
17 }//fim da classe

```

Figura 5.8: Programa que calcula o fatorial de um número qualquer.

O primeiro elemento (1) já está armazenado em $result$, logo, só é necessário fazer as multiplicações de 2 até num . Por isso, a variável de contagem i é inicializada em 2 e finalizada com num na repetição. Como a variável i é inicializada com 2, finalizada com num e incrementada de 1 em 1, ela irá assumir todos os valores necessários para o cálculo do fatorial. Ou seja, na primeira iteração, ela assumirá 2, na segunda 3, na terceira 4 e assim sucessivamente até num . Logo, é possível utilizá-la como termo da multiplicação no cálculo do fatorial.

Para entender melhor este exemplo, acompanhe a tabela a seguir. Nela, estão expostos os valores de cada variável no final de cada iteração para um exemplo em que o usuário informa que quer saber o fatorial de 5:

Tabela 5.3: Valores de cada variável ao fim de cada iteração.

Variável	Antes da Repetição	Iteração 1	Iteração 2	Iteração 3	Iteração 4
num	5	5	5	5	5
i	2	3	4	5	6
result	1	2	6	24	120

Note que a variável *num*, que contém o valor 5, permanece constante. A variável *i*, que é controlada pela estrutura de repetição, aumenta seu valor de 1 em 1. A variável *result* utiliza o valor da variável *i* para multiplicar o próprio valor, fazendo com que contenha o valor do fatorial. Assim, durante a iteração 2, por exemplo, *result* receberá o próprio valor, que é o mesmo do final da iteração 1 (2), multiplicado pelo valor de *i*, que também é o mesmo do final da iteração 1 (3). Logo, no final da iteração 2, *result* receberá o resultado da multiplicação $2 * 3$, que resulta em 6.

Após esta multiplicação, o passo da repetição é executado e o valor de *i* é alterado para 4. Do mesmo modo, na iteração 3, *result* irá receber o resultado da multiplicação $6 * 4$ e assim sucessivamente. O algoritmo para quando *i* atinge 6, uma vez que *num* tem valor 5, e 6 não é menor ou igual a 5.

Combinando estruturas de repetição, desvio e declaração de variáveis

Já vimos que, dentro do corpo de estruturas de repetição, podem existir múltiplas instruções. Alguns casos especiais merecem destaque, como o uso de desvios, repetições e declarações de variáveis dentro de repetições.

Declaração de variáveis

Observe o exemplo da **Figura 5.9**, que expõe dois programas:

<pre> 1 public class VariavelRepeticao1 2 { 3 public static void main(String[] a) 4 { 5 6 int j=0; 7 for(int i=0;i<10;i++) 8 { 9 System.out.println("j="+j++); 10 } //fim da repetição 11 } //fim da função main 12 } //fim da classe </pre>	<pre> 1 public class VariavelRepeticao2 2 { 3 public static void main(String[] a) 4 { 5 6 for(int i=0;i<10;i++) 7 { 8 int j=0; 9 System.out.println("j="+j++); 10 } //fim da repetição 11 } //fim da função main 12 } //fim da classe </pre>
---	---

(a)

(b)

Figura 5.9: Dois programas com estrutura com estrutura for: (a) sem criação de variável dentro da estrutura; (b) com criação de variável dentro da repetição.

Os dois algoritmos estão corretos; entretanto o resultado de cada um será diferente. O programa da **Figura 5.9a** é um algoritmo que imprime números de 0 a 9, já o programa da **Figura 5.9b** imprime 10 linhas com o mesmo valor, que neste caso é 0.

A diferença de resultados ocorre porque, no segundo programa, a variável j é criada e inicializada dentro da repetição, fazendo com que, a cada iteração, uma nova variável j seja criada. É perfeitamente possível criar variáveis dentro de estruturas de repetição, visto que os dois programas estão sintaticamente certos, porém é importante observar que a criação de variáveis dentro das repetições faz com que valores armazenados dentro destas variáveis sejam perdidos. Além disso, uma variável criada dentro de uma repetição só existe dentro da estrutura de repetição, não podendo portanto, ser utilizada em outras partes do código.

Para decidir se devemos criar ou não uma variável dentro de uma estrutura de repetição, temos que analisar o tipo de problema que queremos resolver. Se valores armazenados em variáveis não precisarem ser levados de uma iteração para outra (como no caso da **Figura 5.9b**), estas variáveis poderão ser criadas dentro das repetições. No entanto, se os valores de iterações anteriores são importantes para as iterações seguintes (como nos casos da **Figura 5.9a**, da média aritmética, do fatorial, etc.) estas variáveis devem ser criadas fora da repetição.

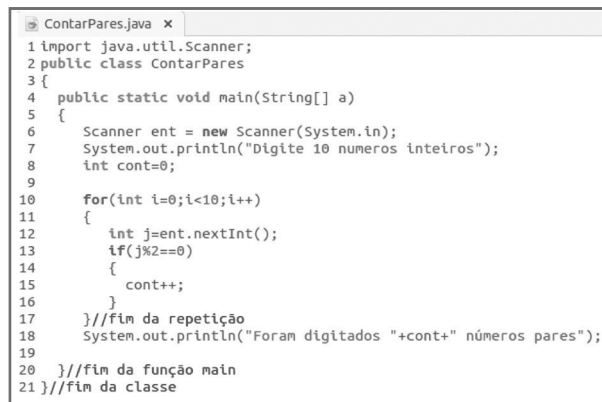
É importante mencionar que variáveis criadas dentro de uma repetição só existem dentro desta repetição. Se mudássemos o exemplo da **Figura 5.9b** para imprimir o valor de j após a linha 10, estaríamos usando uma variável criada dentro de uma repetição fora dela, o que é um erro. Após a repetição, a variável deixa de existir e, ao tentar utilizá-la novamente, o compilador retornará uma mensagem de erro, pois o programa está utilizando uma variável que não existe mais.



Caso você precise criar uma variável para auxiliar em cálculos ou em outras tarefas, e esta variável só for utilizada em uma estrutura (dentro de um desvio ou repetição, por exemplo), recomenda-se que esta variável seja declarada dentro da estrutura. Isto torna o código mais elegante e legível.

Estrutura de desvio

Para combinar estruturas condicionais e de repetição, também vale a regra do aninhamento, a qual diz que, se uma estrutura *a* começa dentro de uma segunda estrutura *b*, então *a* deve terminar antes de *b*. Ou seja, *a* deve estar completamente contida dentro de *b*. Por exemplo: suponha que queiramos ler 10 números digitados pelo usuário e contabilizar quantos destes números são pares. A **Figura 5.10** mostra uma possível solução para este problema:



```
1 import java.util.Scanner;
2 public class ContarPares
3 {
4     public static void main(String[] a)
5     {
6         Scanner ent = new Scanner(System.in);
7         System.out.println("Digite 10 numeros inteiros");
8         int cont=0;
9
10        for(int i=0;i<10;i++)
11        {
12            int j=ent.nextInt();
13            if(j%2==0)
14            {
15                cont++;
16            }
17        } //fim da repetição
18        System.out.println("Foram digitados "+cont+" números pares");
19    } //fim da função main
20 } //fim da classe
```

Figura 5.10: Exemplo de aninhamento de uma estrutura *for* e uma estrutura *if*.

A figura mostra um programa composto por duas estruturas: uma de repetição e uma de desvio. Note que a estrutura de desvio (linhas 13 a 16) começa e termina dentro da estrutura de repetição (linhas 10 a 17).

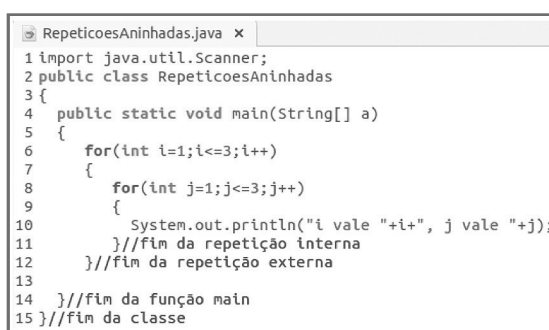
O programa procede da seguinte maneira:

- A variável de contagem *i* é inicializada com o valor 0.
- É verificado se o valor de *i* é menor que 10.
- Como *i* vale 0, o critério de repetição retorna *true* e a repetição prossegue.
- É pedido ao usuário que digite um número.
- Verifica-se se o número digitado é múltiplo de 2, ou seja, se ele é par.
- Caso o número seja par, a variável *cont* é acrescida de uma unidade.
- Como última instrução da repetição, é feito o passo da contagem, fazendo com que *i* assumo o valor 1.
- O programa prossegue até que *i* assumo o valor 10.

- Por último, é impressa uma mensagem para o usuário, dizendo quantos dos números digitados foram pares.

Estrutura de repetição

Considere duas estruturas de repetição: uma estrutura *a*, externa, e uma estrutura *b*, posicionada dentro de *a*. Considere também que *a* está preparada para fazer *n* iterações e *b*, para fazer *m* iterações. Dado o posicionamento destas estruturas, para cada iteração que *a* fizer, *b* fará *m* iterações. A **Figura 5.11** mostra um exemplo de aninhamento entre estruturas de repetição.



```

1 import java.util.Scanner;
2 public class RepeticoesAninhadas
3 {
4     public static void main(String[] a)
5     {
6         for(int i=1;i<=3;i++)
7         {
8             for(int j=1;j<=3;j++)
9             {
10                System.out.println("i vale "+i+", j vale "+j);
11            } //fim da repetição interna
12        } //fim da repetição externa
13    }
14 } //fim da função main
15 } //fim da classe
  
```

Figura 5.11: Exemplo de aninhamento de repetições.

Na **Figura 5.11**, é possível observar que existem duas estruturas de repetição: uma que controla a variável *i* e outra que controla a variável *j*. Também observamos o aninhamento da estrutura que controla *j* em relação à estrutura que controla *i*, ou seja, a estrutura que controla a variável *j* está inteiramente posicionada dentro da estrutura que controla a variável *i*. Esta disposição faz com que, para cada iteração da estrutura externa (a que controla *i*), todas as iterações da estrutura interna (a que controla *j*) sejam executadas. Ou seja, a estrutura externa tem por finalidade repetir a interna. Deste modo, a saída que este algoritmo produz é a seguinte:

```

i vale 1 j vale 1
i vale 1 j vale 2
i vale 1 j vale 3
i vale 2 j vale 1
  
```

i vale 2 j vale 2

i vale 2 j vale 3

i vale 3 j vale 1

i vale 3 j vale 2

i vale 3 j vale 3

Atividade 3

Atende aos objetivos 1, 2 e 3

- Faça um programa que leia 10 números informados pelo usuário e conte quantos destes foram pares e quantos destes foram ímpares.
- Faça um programa que exiba na tela a seguinte imagem:

```
*
* * *
* * * * *
* * * * * *
* * * * * * *
```

- Faça um programa para determinar se um número qualquer, informado pelo usuário, é primo.
- Faça um programa para resolver a seguinte série:

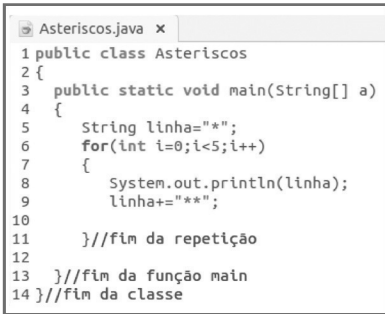
$$\text{sen}(x) = x - x^3/3! + x^5/5! - x^7/7! \dots$$

Nesta série, x é um ângulo dado em radianos e o número de termos utilizados deve ser informado pelo usuário.

Resposta comentada

a) Para resolver esta questão, basta modificar o programa da **Figura 5.10**. Deve ser criado mais um contador (para os números ímpares). A instrução que modifica o contador dos ímpares deve ser colocada dentro de um *else*, para que a contagem dos pares seja separada da dos ímpares. No final, os dois contadores devem ser impressos.

b) Para resolver esta questão, basta analisar o comportamento das linhas da figura a seguir. São apresentadas cinco linhas, a primeira com um asterisco, a segunda com três asteriscos, a terceira com cinco asteriscos, a quarta com sete asteriscos e a quinta com nove asteriscos. Note que o que é mostrado na tela é um texto (conjunto de asteriscos) e que o número de asteriscos aumenta em duas unidades para cada linha. Uma estratégia simples utilizando concatenação de textos pode resolver o problema. Esta solução é apresentada na **Figura 5.12**:

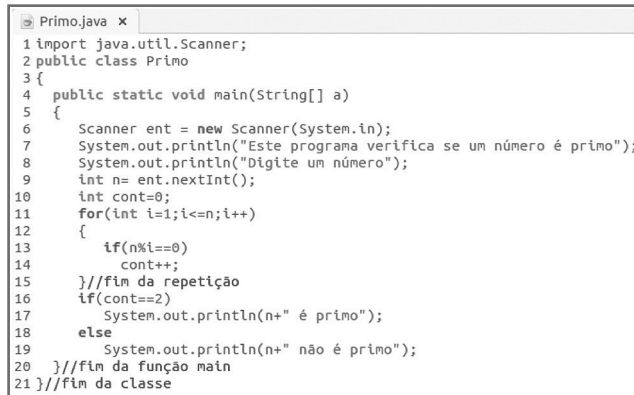


```
1 public class Asteriscos
2 {
3     public static void main(String[] a)
4     {
5         String linha="*";
6         for(int i=0;i<5;i++)
7         {
8             System.out.println(linha);
9             linha+="**";
10        }
11    }
12 }
13 //fim da função main
14 //fim da classe
```

Figura 5.12: Solução utilizando concatenação de textos.

c) Para resolver esta questão, primeiro é preciso relembrar o conceito de número primo. Um número primo é um número que só possui dois divisores: 1 e ele próprio. Baseado nesta propriedade, podemos utilizar um algoritmo simples, para determinar se um número é primo ou não, que utiliza a estratégia de contar quantos divisores este número tem. Se existirem mais que dois divisores, este número não é primo. A solução é mostrada na **Figura 5.13**.

Na solução apresentada, a variável n é o número que será verificado. A variável i controla os valores dos possíveis divisores e a variável $cont$ é responsável por armazenar quantos foram os divisores de n .



```

1 import java.util.Scanner;
2 public class Primo
3 {
4     public static void main(String[] a)
5     {
6         Scanner ent = new Scanner(System.in);
7         System.out.println("Este programa verifica se um número é primo");
8         System.out.println("Digite um número");
9         int n = ent.nextInt();
10        int cont=0;
11        for(int i=1;i<=n;i++)
12        {
13            if(n%i==0)
14                cont++;
15        }//fim da repetição
16        if(cont==2)
17            System.out.println(n+" é primo");
18        else
19            System.out.println(n+" não é primo");
20    }//fim da função main
21 }//fim da classe

```

Figura 5.13: Se existirem mais que dois divisores, este número não é primo.

Obs.: Existem vários modos para tornar este programa mais eficiente. Contudo, por questões didáticas, a eficiência é um tópico que não será abordado neste momento.

d) Nesta série, x é um ângulo dado em radianos, e o número de termos utilizados deve ser informado pelo usuário. Para resolver este problema, primeiro é preciso entender sua notação. As reticências no final da equação indicam que esta é uma série infinita, ou seja, tem um número de termos infinito. Como não é correto fazer um algoritmo com infinitos passos, devemos estabelecer um critério de parada para o cálculo da série.

O próprio enunciado já determina que o critério de parada seja o número de termos informado pelo usuário. Um termo da série é compreendido como um valor e um sinal. Assim, o primeiro termo desta série é $+x$. o segundo é $-x^3/3!$. O terceiro é $+x^5/5!$, e assim sucessivamente. Deste modo, se o usuário determinar que o cálculo da série deve ser feito com dois termos, então retornaremos o resultado de $x - x^3/3!$.

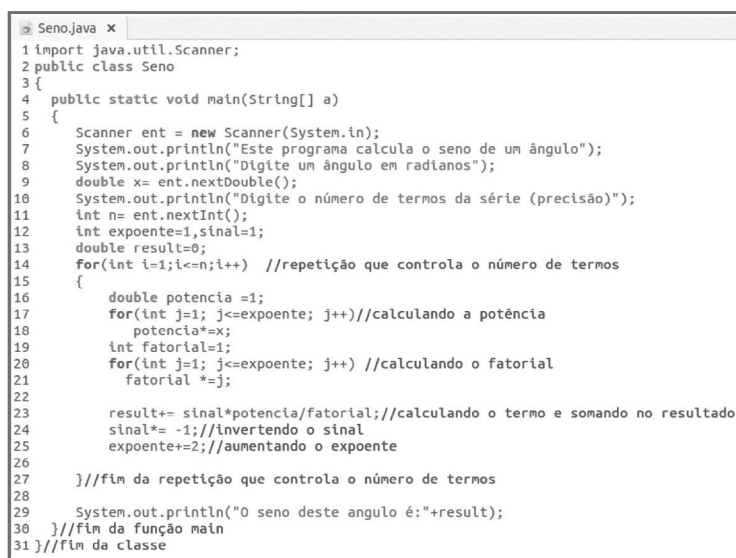
Ainda sobre os termos da série, é importante mencionar que o primeiro termo é $x = x^1/1!$ e que, a cada termo, ocorrem duas mudanças:

- o expoente do ângulo e o parâmetro do fatorial no denominador são acrescidos de duas unidades;
- o sinal do termo é invertido.

Assim, para obtermos o quinto termo, utilizamos como base o quarto, aumentamos o expoente do ângulo e o parâmetro do fatorial no denominador em duas unidades e invertemos o sinal. Como o quarto termo é $-x^7/7!$, o quinto fica $+x^9/9!$, e assim sucessivamente.

Por último, note que a série é apresentada como $\text{sen}(x) = \text{série}$. Este tipo de notação significa que o valor do $\text{sen}(x)$ é aproximado pela série. Logo, o valor do $\text{sen}(x)$ será encontrado através da série, e não o contrário. Também é importante mencionar que, quanto maior o número de termos da série, mais próximo do valor do $\text{sen}(x)$ o resultado estará.

Tendo em vista estas explicações, uma possível solução é apresentada na imagem a seguir, onde a resposta final é calculada termo a termo, começando em $x^1/1!$ e indo até o enésimo termo.



```

1 import java.util.Scanner;
2 public class Seno
3 {
4     public static void main(String[] a)
5     {
6         Scanner ent = new Scanner(System.in);
7         System.out.println("Este programa calcula o seno de um ângulo");
8         System.out.println("Digite um ângulo em radianos");
9         double x= ent.nextDouble();
10        System.out.println("Digite o número de termos da série (precisão)");
11        int n= ent.nextInt();
12        int expoente=1,sinal=1;
13        double result=0;
14        for(int i=1;i<=n;i++) //repetição que controla o número de termos
15        {
16            double potencia =1;
17            for(int j=1; j<=expoente; j++)//calculando a potência
18                potencia*=x;
19            int fatorial=1;
20            for(int j=1; j<=expoente; j++) //calculando o fatorial
21                fatorial *=j;
22
23            result+= sinal*potencia/fatorial;//calculando o termo e somando no resultado
24            sinal*= -1;//invertendo o sinal
25            expoente+=2;//aumentando o expoente
26        }
27        //fim da repetição que controla o número de termos
28        System.out.println("O seno deste angulo é:"+result);
29    }
30 }
31 }

```

Figura 5.14: Resposta calculada termo a termo, começando em $x^1/1!$ e indo até o enésimo termo.

A variável *expoente* é utilizada para armazenar o valor do expoente do ângulo e do parâmetro do fatorial no termo atual. Como estes valores são acrescidos de dois em dois na série, o programa também deve fazer isso. Para fazer a inversão de sinal, a solução apresentada utiliza uma variável que é multiplicada pelo valor do termo. Como a cada termo o sinal é invertido, esta variável é multiplicada por -1 , o que faz com que seu valor se alterne entre 1 e -1 .

Obs.: Devido ao fato de as séries serem aproximações da função original, para qualquer número de termos diferente de infinito, o que se obtém

com este programa é uma aproximação do resultado da função seno. Além disso, devido à limitação dos computadores modernos, este programa, concebido como está, não pode ser utilizado para um número grande de termos. Isto ocorre porque os computadores possuem limites de valores que podem ser representados numericamente. Alguns destes limites serão estudados nesta disciplina em momento oportuno, outros você aprenderá como calcular em outras disciplinas. Por enquanto, para testar seu programa não use números de termos maiores que 6.

Resumo

Nesta aula, você aprendeu a lidar com estruturas de repetição com contador e a compor programas com aninhamento de estruturas de desvio e repetição. Observamos que a quantidade de repetições é determinada pela combinação de três componentes da estrutura de repetição (inicialização, critério de repetição e passo) e que, quando mal-elaborada, esta combinação pode levar à construção de algoritmos/programas mal formados. Também estudamos a criação de variáveis dentro de estruturas de repetição e mostramos que este tipo de variável não pode ser utilizada fora da estrutura.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com a repetição de tarefas quando não é possível determinar o número de vezes que a tarefa será repetida.

Referências

ASCENCIO, A. F. G.; Campos, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

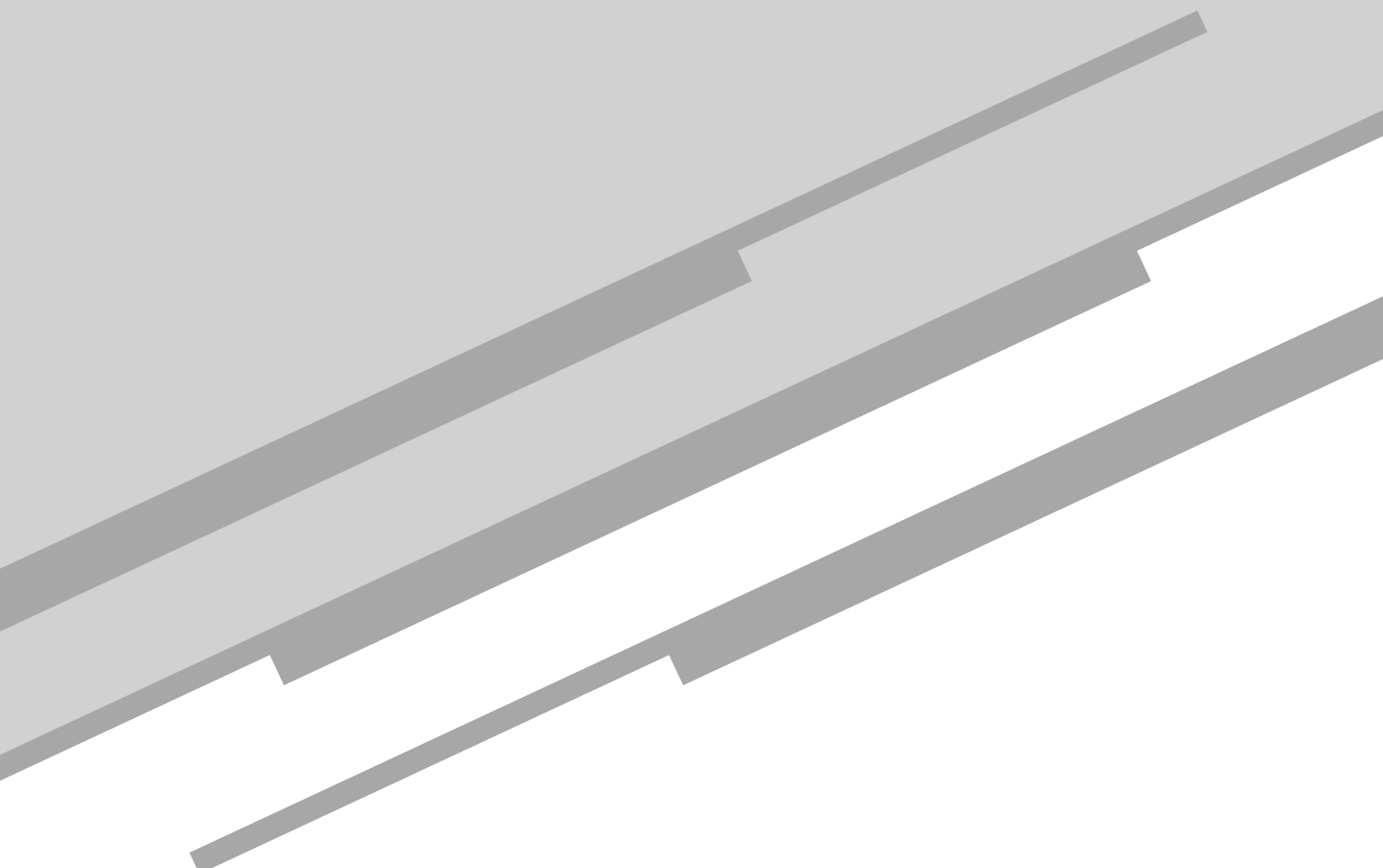
CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. et al. *Programação estruturada de computadores*. 2^a ed. Rio de Janeiro: Guanabara, 1989.

Aula 6

Repetição – Parte II



Meta

Expor os conceitos de repetição sem contador e as duas estruturas usadas para este fim.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. aplicar estruturas de repetição sem contador;
2. fazer programas que repetição sem contador para resolver problemas.

Introdução

Até agora, você aprendeu que, para construir programas que repetem múltiplas vezes uma mesma tarefa, é necessário fazer uso de estruturas de repetição. Entretanto, foram apresentados somente casos em que é possível determinar o número de vezes que a tarefa será repetida. Existem casos nos quais não é possível determinar o número de vezes que uma mesma tarefa será repetida e, nestas situações, o uso de uma estrutura de repetição com contador se torna inadequado.

Exemplo: Faça um programa que leia números positivos que o usuário digite e os imprima na tela. O algoritmo deve parar sua execução quando a soma dos números digitados for maior ou igual a 10. Este é um caso simples, em que não é adequado utilizar uma estrutura de repetição com contador. Para usar a estrutura com contador, seria necessário perguntar ao usuário quantos números no total ele quer digitar. Entretanto, nem sempre o usuário sabe, *a priori*, quantas vezes realizará uma tarefa. Um exemplo típico é a compra de itens pela Internet. Nem sempre o usuário compra todos os itens que planejava comprar, ou ainda é possível que ele aproveite alguma promoção e até compre mais do que planejava. Supor um número fixo de itens de compra, neste caso, pode não ser uma boa estratégia.



Figura 6.1: Um dos modos de matar alguém usando a Internet.

Com estes exemplos, fica claro que repetição com contadores é uma opção que traz certas limitações. Em alguns casos, é necessário repetir tarefas sem pressupor o número final de repetições. Para estes casos, usaremos uma estrutura de repetição sem contador.



Código eficiente

É importante mencionar que tudo o que pode ser feito com uma estrutura de repetição com contador também pode ser feito com uma estrutura de repetição sem contador.

Então surge uma pergunta: Por que estudar dois tipos de estrutura de repetição, se a estrutura de repetição sem contador resolve todos os casos? A resposta é eficiência. As estruturas de repetição com contador podem ser otimizadas pelos compiladores de algumas linguagens para que executem mais rapidamente nos computadores. Além disso, para problemas com necessidade de contadores, a estrutura com contador proporciona programas mais simples.

Nesta aula, você aprenderá a lidar com duas estruturas de repetição sem contador. Também aprenderá a combinar estruturas com/sem contador, juntamente com estruturas de desvio condicional para compor programas. Além disso, veremos casos em que será analisada a necessidade de estruturas sem contador e quando devemos aplicar cada uma delas.

Estrutura de repetição sem contador e controle no início

A **Figura 6.2** mostra a sintaxe da estrutura *while*. Este tipo de repetição sempre começa com a palavra reservada *while* seguida de um par de parênteses que envolvem a condição de repetição, seguido de uma instrução ou de um bloco de instruções. A semântica desta estrutura é a mesma da estrutura *Enquanto* usada em algoritmos vistos na disciplina Computação I.

A instrução (ou bloco de instruções) é repetida enquanto a condição de repetição for *true*. Quando a condição for avaliada como *false*, a repetição é encerrada.

Note que uma das instruções dentro da estrutura de repetição deve cuidar para que, em algum momento, a expressão lógica resulte em falso, pois, caso contrário, a expressão sempre resultará em verdadeiro e, consequentemente, a estrutura de repetição não será interrompida.

```
while(<condição de repetição>)
<instrução a repetir>

while(<condição de repetição>)
{
<conjunto de instruções a repetir>
}
```

Figura 6.2: Sintaxe da estrutura *while*.

Agora que você conhece a estrutura *while*, vejamos seu uso em um exemplo simples. A **Figura 6.3** mostra o uso da estrutura *while* para resolver um dos problemas expostos na introdução desta aula. Faça um programa que leia números positivos digitados pelo usuário e os imprima na tela. O algoritmo deve parar sua execução quando a soma dos números digitados for maior ou igual a 10.

```
testeWhile.java x
1 import java.util.Scanner;
2 public class testeWhile
3 {
4     public static void main(String[] a)
5     {
6
7         double soma=0;//declaração de variáveis
8         Scanner ent = new Scanner(System.in);
9
10        while(soma<10)//repetição sem contador
11        {
12            System.out.println("digite um número");
13            soma+= ent.nextDouble();
14        }//fim da repetição
15        System.out.println("A soma dos números digitados atingiu 10");
16    }//fim da função main
17 }//fim da classe
```

Figura 6.3: Exemplo de uso da estrutura *Enquanto*.

O programa procede da seguinte maneira: no início, cria-se uma variável *double soma*. Neste exemplo, *soma* é utilizada para armazenar a soma dos números lidos. Em seguida, é avaliada a condição. Caso a expressão lógica resulte em verdadeiro, ou seja, se a soma for menor que 10, as instruções dentro da repetição serão executadas. Caso contrário, o fluxo é desviado para o fim da repetição (linha 14) e a mensagem “A soma dos números digitados atingiu 10” é exibida e o programa é encerrado. Dentro da repetição, é feita uma interação com o usuário juntamente com a leitura de um número. O número lido é adicionado ao valor da variável *soma* (linha 13). Após esta operação, a condição é

avaliada novamente. Caso ela continue verdadeira, as instruções serão repetidas novamente. Caso ela se torne falsa, o fluxo é desviado para o final da repetição (linha 14), a mensagem da linha 15 é exibida e o programa termina. Note que a condição de parada é expressa em função do valor de *soma*. Isso faz com que as instruções dentro da repetição sejam executadas enquanto o valor de *soma* for menor ou igual a 10.

Estrutura de repetição sem contador e controle no fim

A **Figura 6.4** mostra a sintaxe da estrutura de repetição *do-while*. Esta estrutura começa com a palavra reservada *do* e termina com *while*, seguido de uma expressão lógica. A semântica é similar à da estrutura *while*. As instruções dentro da estrutura são repetidas enquanto a expressão lógica resultar em verdadeiro. A diferença entre as duas estruturas é que as instruções da estrutura *do-while* são executadas pelo menos uma vez. Ou seja, independentemente do valor da expressão, as instruções são executadas na primeira vez. Dependendo do valor da expressão, haverá uma segunda iteração e, assim, sucessivamente. Observe também que o ponto e vírgula (;) faz parte da sintaxe obrigatória desta estrutura.

```
do
    <instrução a repetir>
while(<condição>);

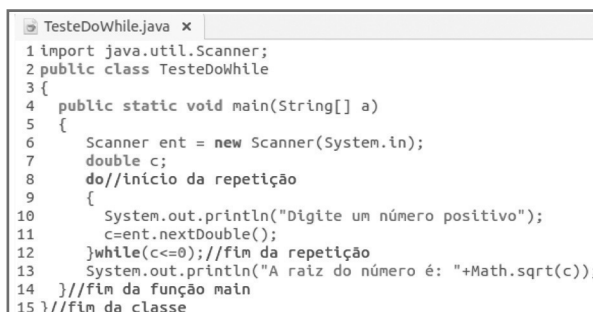
do
{
    <instruções a repetir>
}while(<condição>);
```

Figura 6.4: Sintaxe da estrutura *do-while*.

É importante mencionar que as duas estruturas (*do-while* e *while*) podem ser intercambiadas com as devidas adaptações. Existem casos nos quais, por simplicidade, é mais adequado utilizar uma ou outra estrutura.

Exemplo: Faça um programa para calcular a raiz quadrada de um número real positivo. O algoritmo deve se certificar de que o usuário digitará um número positivo antes de calcular a raiz. A **Figura 6.5** mostra uma possível solução para o problema. Note que fazer algo similar com a estrutura *while* requer pelo menos uma instrução a mais, que seria uma inicialização da variável *c*. Neste exemplo em particular, qualquer instrução extra, implica um aumento de aproximadamente 8% no número de instruções do programa. Como em alguns casos o número de instruções do programa (tamanho) tem impacto direto no tempo de

execução (programas que cabem dentro da memória cache dos computadores executam muito mais rápido), usar mais instruções que o necessário pode não ser vantajoso.



```

1 import java.util.Scanner;
2 public class TesteDoWhile
3 {
4     public static void main(String[] a)
5     {
6         Scanner ent = new Scanner(System.in);
7         double c;
8         do//inicio da repetição
9         {
10             System.out.println("Digite um número positivo");
11             c=ent.nextDouble();
12         }while(c<=0);//fim da repetição
13         System.out.println("A raiz do número é: "+Math.sqrt(c));
14     }//fim da função main
15 }//fim da classe
  
```

Figura 6.5: Exemplo de uso da estrutura *do-while*.

Vale a pena ressaltar que eficiência de algoritmos não é o foco desta disciplina. O objetivo, neste momento, é aprender a compor programas corretos, ou seja, capazes de fornecer respostas corretas para os problemas abordados. Neste sentido, você pode utilizar a estrutura que preferir, desde que seu programa esteja correto.

Econtrando limites numéricos: um exemplo de aplicação da repetição sem contadores

Diferentemente da matemática, o conjunto de números inteiros que os computadores conseguem representar é finito. Em Java, este conjunto possui dois limites: o menor, aqui chamado a , é negativo, e o maior, aqui chamado b , é positivo. Todos os números no intervalo $[a,b]$ são representados, porém Java não consegue trabalhar com números inteiros fora deste intervalo. Este conjunto finito de números inteiros é representado internamente de maneira circular, e não na forma de reta como tradicionalmente feito pela matemática. Isto significa que, a soma de dois números positivos pode resultar em um número negativo, se o resultado for um número maior que b . A **Figura 6.6** irá ajudar a entender esta representação.

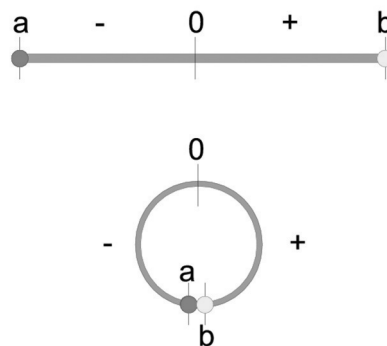
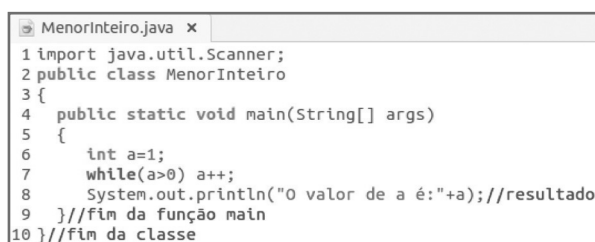


Figura 6.6: Representação do sistema numérico de números inteiros. Acima, modo tradicional. Abaixo, modo utilizado por várias linguagens de programação.

Nesta figura o sistema numérico dos números inteiros está representado da maneira tradicional, em forma de reta, e da maneira como é implementado em várias linguagens de programação (Java entre elas), na forma de circunferência.

Imagine a seguinte situação: pegue o valor $b - 1$ e some 3. Matematicamente, o resultado é $b + 2$, porém este número não pode ser representado. Como nesta representação a e b são vizinhos, isto implica $b + 1 = a$, e $a - 1 = b$. Logo $b + 2 = a + 1$, sendo portanto o resultado $(b - 1) + 3 = a + 1$, que é um número negativo. É importante mencionar que a e b não têm o mesmo valor absoluto. Usando estas informações, é possível fazer um programa que determine o menor valor inteiro do tipo *int* que Java pode representar.

Apesar de haver muitos conceitos diferentes envolvidos neste tópico, o programa para solucionar o problema é bem simples. Se soubermos o valor de b , encontramos a através da equação $a = b + 1$. Porém não sabemos *a priori*, os valores de a e b . Entretanto, é fácil perceber que, se partirmos de um número positivo qualquer e a este número adicionarmos o valor 1 sucessivas vezes, em um determinado momento teremos o valor de b . Neste ponto, se somarmos 1 novamente, teremos a . O ponto chave da questão é descobrir o mecanismo de parada desta repetição. Note que apesar de serem vizinhos nesta representação circular, a e b têm sinais opostos. Portanto, tudo o que se deve fazer é pegar um número inteiro positivo qualquer, e a este número adicionar 1 sucessivas vezes, até que o resultado mude de sinal. Neste momento teremos exatamente o valor de a . A seguir, uma possível solução para este problema:



```

MenorInteiro.java x
1 import java.util.Scanner;
2 public class MenorInteiro
3 {
4     public static void main(String[] args)
5     {
6         int a=1;
7         while(a>0) a++;
8         System.out.println("O valor de a é:"+a);//resultado
9     }//fim da função main
10 }//fim da classe

```

Figura 6.7: Partindo de um número positivo qualquer e a este número adicionando o valor 1 sucessivas vezes, em determinado momento teremos o valor de b, ao qual, se somarmos 1 novamente, teremos a.

Atividade 1

Atende aos objetivos 1 e 2

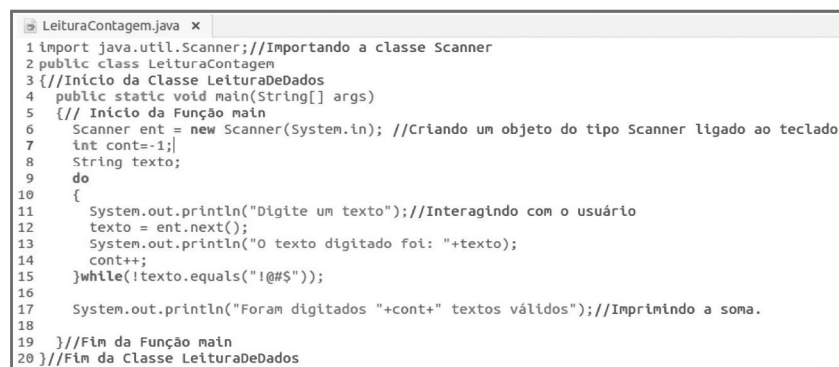
a) Faça um programa que leia vários textos e os imprima na tela. O programa deve parar a leitura quando for digitado o texto “!@#\$.” Também deve ser calculado e mostrado quantos textos foram digitados. Obs.: O texto “!@#\$.” não deve ser contabilizado.

b) Faça um programa que calcule a média aritmética de uma turma em que o número de alunos não é informado. O algoritmo deve parar de calcular a média quando uma nota negativa for digitada.

c) Um determinado material radioativo perde metade de sua massa a cada x segundos. Faça um programa para calcular o tempo, em horas, minutos e segundos, para que uma determinada massa y deste material seja menor ou igual a um determinado limiar de massa z , sendo $y > z$. Os parâmetros x , y , z devem ser informados pelo usuário.

Resposta comentada

a) Para resolver esta questão, devemos utilizar uma variável de contagem em conjunto com uma estrutura de repetição sem contador, visto que a condição é imprimir textos até que um determinado texto seja digitado. Como pelo menos um texto deve ser digitado, a melhor opção é utilizar a estrutura *do-while*. Para realizar a contagem, podemos somar 1 a uma variável todas as vezes que um texto for digitado. Entretanto, isso levará à contabilização do texto “!@#”, o que o enunciado expressamente diz que não deve ser feito. Para resolver este problema, ao final da repetição, deve-se subtrair o valor da variável de contagem em uma unidade, removendo assim, o valor contabilizado pela digitação do texto “!@#”. Outra possível solução é começar a contagem a partir de -1 ao invés de 0. Isso fará com que a contagem de textos seja defasada em uma unidade.



```
1 import java.util.Scanner; //Importando a classe Scanner
2 public class LeituraContagem
3 { //Início da Classe LeituraDeDados
4     public static void main(String[] args)
5     { // Início da Função main
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner ligado ao teclado
7         int cont=-1;
8         String texto;
9         do
10         {
11             System.out.println("Digite um texto");//Interagindo com o usuário
12             texto = ent.next();
13             System.out.println("O texto digitado foi: "+texto);
14             cont++;
15         }while(!texto.equals("!@#"));
16
17         System.out.println("Foram digitados "+cont+" textos válidos");//Imprimindo a soma.
18
19     } //Fim da Função main
20 } //Fim da Classe LeituraDeDados
```

Figura 6.8: Possível implementação da solução expressa anteriormente.

Outra solução para este mesmo problema utiliza aninhamento de uma estrutura de desvio condicional (linhas 13 a 17) dentro de uma estrutura de repetição (linhas 9 a 18). Note que, além das diferenças óbvias referentes à inserção da estrutura de desvio condicional, também há diferenças sutis na inicialização das variáveis.

```

LeituraContagem2.java x
1 import java.util.Scanner; //Importando a classe Scanner
2 public class LeituraContagem2
3 { //Início da Classe LeituraDeDados
4     public static void main(String[] args)
5     { // Início da Função main
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner ligado ao teclado
7         int cont=0;
8         String texto="";
9         while(!texto.equals("!@#$"))
10        {
11            System.out.println("Digite um texto");//Interagindo com o usuário
12            texto = ent.next();
13            if(!texto.equals("!@#$"))
14            {
15                System.out.println("O texto digitado foi: "+texto);
16                cont++;
17            }
18        }
19        System.out.println("Foram digitados "+cont+" textos válidos");//Imprimindo a soma.
20    } //Fim da Função main
21 } //Fim da Classe LeituraDeDados

```

Figura 6.9: Implementação alternativa utilizando aninhamento de estruturas para o problema.

b) Para resolver esta questão, será usada uma estratégia diferente:

```

MediaEnquanto.java x
1 import java.util.Scanner; //Importando a classe Scanner
2 public class MediaEnquanto
3 { //Início da Classe LeituraDeDados
4     public static void main(String[] args)
5     { // Início da Função main
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner ligado ao teclado
7         int cont=0;
8         double nota,soma=0;
9         do
10        {
11            System.out.println("Digite uma nota");//Interagindo com o usuário
12            nota = ent.nextDouble();
13            soma+=nota;
14            cont++;
15        }while(nota>=0);
16        cont--;
17        soma/=nota;
18        System.out.println("A média da turma é: "+soma/cont);//Imprimindo a soma.
19    } //Fim da Função main
20 } //Fim da Classe LeituraDeDados

```

Figura 6.10: Possível solução para a questão.

Nesta solução, são criadas: uma variável *cont* para contabilizar quantos alunos tem a turma, uma variável *soma* para calcular a soma das notas e uma variável *nota* para lidar com as leituras de cada nota, individualmente. Inicialmente, o valor zero é atribuído às variáveis *soma* e *cont*. Em seguida, é feita uma repetição, que é responsável por ler cada uma das notas e atualizar a soma e o número de alunos. Note que, mesmo quando for digitada uma nota negativa, esta nota é contabilizada em soma e é acrescida uma unidade na variável *cont*. Isto é intencional, pois a estratégia utilizada é corrigir o valor destas variáveis, posteriormente.

Logo após a repetição, a variável *cont* é decrescida de uma unidade. Isto é feito para corrigir o valor desta variável. Note que, ao digitar uma nota negativa, *cont* também é aumentada em uma unidade pela instrução da

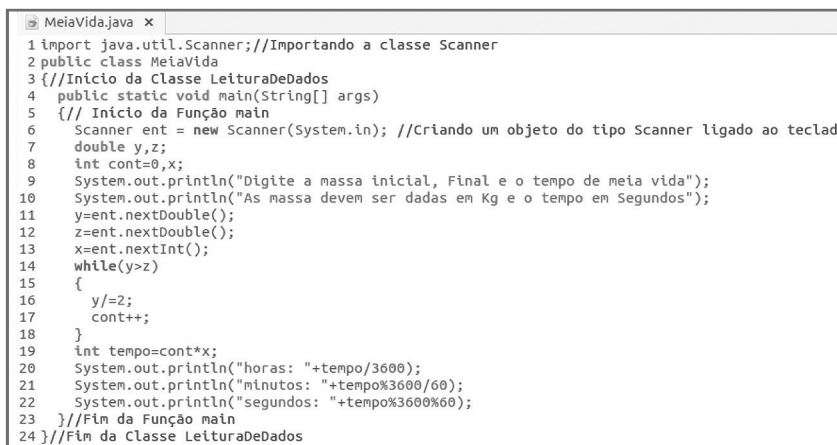
linha 14. Contudo, a instrução da linha 15 não deixa que a repetição prossiga. Isto significa que, ao sair da repetição, *cont* estará sempre com uma unidade a mais do que deveria. Sendo assim, basta subtrair uma unidade para corrigir seu valor. Um acerto similar é feito para a variável *soma*. Note que a nota negativa digitada é incorporada em *soma* pela instrução da linha 13. Mas a instrução da linha 15 não deixa que a repetição prossiga. Logo, o que deve ser feito é subtrair aquele valor negativo incorporado, para corrigir o valor de *soma*. Por último, a média é calculada e exibida.

c) A primeira coisa a ser feita é perceber que, apesar de a meia vida de materiais radioativos poder ser descrita como uma progressão geométrica (PG), aplicar as fórmulas de progressão pode não dar bons resultados. Note que, apesar de *y* sempre ser o termo inicial da PG de razão meio, não necessariamente *z* é parte da progressão. Exemplo: suponha que um material perde metade de sua massa a cada 45 segundos e queremos saber quanto tempo é necessário para que 10 kg deste material atinjam um limiar de 4 kg. Note que a progressão seria 10, 5, 2,5, ... Quando o limiar é um termo da progressão, fica trivial encontrar a quantidade de termos. Por exemplo: se o limiar fosse 2,5, bastaria utilizar a equação geral do termo da progressão dada por $a_n = a_1 q^{n-1}$, onde $a_n = 2.5$, $a_1 = 10$ e $q = 1/2$ para encontrar o valor de *n*.

O problema é quando o limiar desejado está entre os termos da progressão, como é o caso do exemplo, onde o limiar é 4. Isso inviabiliza o uso das expressões da progressão para resolver o problema, visto que o limiar desejado não é um termo da progressão. Deste modo, a solução envolve calcular os termos da progressão até encontrar o primeiro termo menor ou igual ao limiar, calculando, assim, o número de meias vidas necessárias para chegar até lá.

Uma vez que temos o número de meias vidas, basta multiplicá-lo pelo tempo de cada meia vida para obtermos o tempo total do processo, em segundos. No exemplo mencionado, note que são necessárias duas meias vidas para atingir o limiar. Como cada meia vida leva 45 segundos, tem-se um total de 90 segundos. Agora, basta transformar isso em horas minutos e segundos, ou seja, 0h1min30s. Para fazer isso, basta encontrar os quocientes e os restos de divisões apropriados. Por exemplo, uma hora tem 3.600 segundos, assim, para saber quantas horas o processo levou, basta encontrar o quociente do tempo por 3.600. Note que o resto desta divisão (tempo % 3.600) apresenta os segundos, já excluídas as horas. Basta, então, transformar este resto em minutos, utilizando como divisor o valor 60. Um processo análogo deve ser feito para

encontrar os segundos. Uma possível solução é apresentada a seguir. Nesta solução, a variável *cont* é utilizada para contar o número de meias vidas. A variável *x* armazena o tempo da meia vida e as variáveis *y* e *z* são a massa inicial e o limiar de massa, respectivamente.



```

1 import java.util.Scanner; //Importando a classe Scanner
2 public class MeiaVida
3 { //Início da Classe LeituraDeDados
4     public static void main(String[] args)
5     { // Início da Função main
6         Scanner ent = new Scanner(System.in); //Criando um objeto do tipo Scanner ligado ao teclado
7         double y,z;
8         int cont=0,x;
9         System.out.println("Digite a massa inicial, Final e o tempo de meia vida");
10        System.out.println("As massa devem ser dadas em Kg e o tempo em Segundos");
11        y=ent.nextDouble();
12        z=ent.nextDouble();
13        x=ent.nextInt();
14        while(y>z)
15        {
16            y/=2;
17            cont++;
18        }
19        int tempo=cont*x;
20        System.out.println("horas: "+tempo/3600);
21        System.out.println("minutos: "+tempo%3600/60);
22        System.out.println("segundos: "+tempo%3600%60);
23    } //Fim da Função main
24 } //Fim da Classe LeituraDeDados

```

Figura 6.11: Possível solução para a questão.

Note que, uma vez calculado o tempo total do processo em segundos, há várias formas de fazer a transformação do tempo. Uma delas é ir subtraindo o valor total e calcular as horas e minutos através de estruturas de repetição.

Resumo

Nesta aula você aprendeu a lidar com estruturas de repetição sem contador e a compor algoritmos com aninhamento de estruturas de desvio e repetição. Foi visto que a quantidade de repetições nem sempre pode ser determinada e que, nestes casos, utilizar estruturas de repetição sem contador é mais adequado.

Os objetivos da aula foram contemplados ao longo das seções, onde exemplos de aplicação de estruturas de repetição sem contador foram discutidos, e por meio de atividades que utilizaram estas estruturas na resolução de problemas matemáticos e do cotidiano.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com armazenamento de dados em vetores e matrizes.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

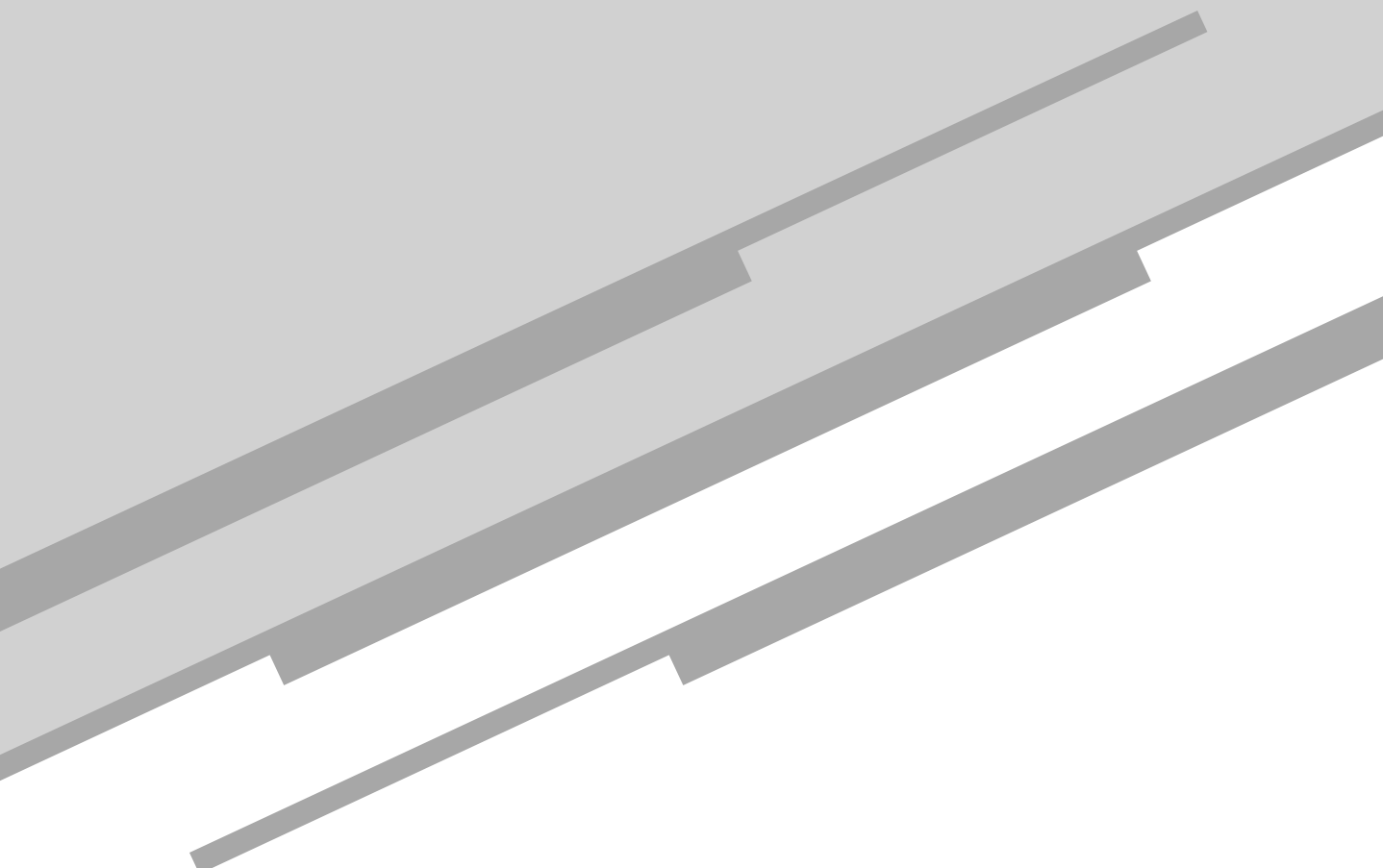
CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. et al. *Programação estruturada de computadores*. 2^a ed. Rio de Janeiro: Guanabara, 1989.

Aula 7

Vetores e matrizes



Meta

Expor os conceitos de vetores e matrizes, mostrando sua sintaxe e funcionamento na linguagem Java.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. definir e aplicar o conceito de vetores e matrizes;
2. fazer programas que combinem estruturas de desvio e repetição juntamente com vetores/matrizes para resolver problemas.

Introdução

Há uma célebre frase de David Hume que diz: “O papel principal da memória é conservar não simplesmente as ideias, mas sua ordem e sua posição”. Mal sabia ele que, cerca de 250 anos após seu nascimento, seriam criadas máquinas com componentes chamados de “memória”, nos quais as informações seriam armazenadas e indexadas por sua posição.

O armazenamento de informações é vital para praticamente toda a ciência. Este armazenamento pode ser feito de modo temporário (anotações, rascunhos, etc.) ou em modo mais permanente (livros, artigos, etc.). Na computação não é diferente. O uso de informações armazenadas é essencial para a resolução de alguns problemas.

Um exemplo real com que muitos engenheiros lidam no dia a dia é o cálculo de **variância**.

A variância pode ser dada por:

$$\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2$$

onde μ (lê-se “mi”) é a média do conjunto de medidas que queremos analisar e n é a quantidade de elementos desse conjunto.

Programas para resolver partes desta fórmula já foram feitos em aulas anteriores. O cálculo de média já foi objeto de nosso estudo. Mas este é apenas o começo do cálculo de variância.

Um exemplo pode ajudar a compreender a complexidade da operação:

A partir de um conjunto numérico

$$x = \{1, 2, 3, 4, 5, 6\}$$

- calcular a média do conjunto (μ):

$$(1 + 2 + 3 + 4 + 5 + 6) / 6 = 3.5,$$

- subtrair essa média de cada um dos elementos utilizados no cálculo dela (por conveniência, cria-se um segundo conjunto):

$$x' = \{(1-3.5), (2-3.5), (3-3.5), (4-3.5), (5-3.5), (6-3.5)\}$$

$$x' = \{-2.5, -1.5, -0.5, 0.5, 1.5, 2.5\},$$

- elevar o resultado dessas subtrações ao quadrado, criando um terceiro conjunto:

Variância

Medida de dispersão utilizada em estatística. Ela indica o quão longe, em geral, os valores de uma determinada medida se encontram do valor esperado dela.

$$x'' = \{(-2.5)^2, (-1.5)^2, (0.5)^2, (0.5)^2, (1.5)^2, (2.5)^2\}$$

$$x'' = \{6.25, 2.25, 0.25, 0.25, 2.25, 6.25\},$$

- somar os quadrados:

$$6.25 + 2.25 + 0.25 + 0.25 + 2.25 + 6.25 = 17.5$$

- e dividir esse total pelo número de elementos do conjunto (n).

$$17.5 / 6 = 2.9166666...$$

Ou seja, a variância do conjunto {1, 2, 3, 4, 5, 6} é algo próximo de 2.91. Note que, sem o armazenamento dos números do conjunto, resolver este problema seria impossível. Poderíamos construir um programa que implementasse apenas os dois primeiros passos do algoritmo acima – o que corresponde ao cálculo da média –, mas, sem um mecanismo para armazenar os valores do conjunto numérico, não seria possível implementar os demais passos, e portanto, o programa ficaria incompleto.

Outra analogia que podemos aplicar para entender a necessidade do armazenamento de dados é quando temos tantos compromissos para fazer em um determinado dia que fica impossível lembrar de todos sem o auxílio de algum dispositivo, como agenda, celular, etc. Ou seja, em muitos casos, não poder armazenar informações de alguma forma torna extremamente difícil, ou mesmo impossível, a realização de algumas tarefas. De modo semelhante, de nada adianta ter as informações armazenadas sem que seja possível acessá-las. Imagine qual seria a utilidade de uma agenda em que se pudesse inserir informações, mas em que não se pudesse consultá-las.

Nesta aula, você aprenderá a lidar com vetores e matrizes, que são um dos modos de armazenar informações em algoritmos e programas.

Vetores: declaração e acesso

Vetores, *arrays* ou matrizes unidimensionais são sinônimos dentro do mundo da informática. Fisicamente, são um conjunto identificável de posições de memória, contíguas, nas quais se podem armazenar informações de um tipo específico. Podem ser vistos como um conjunto de elementos do mesmo tipo em que se pode identificar, acessar e alterar cada um desses elementos.

Dois exemplos de declaração de *arrays* são dados a seguir:

```
double [ ] notas;
int [ ] idades, faltas;
```

A sintaxe da primeira linha é utilizada para declarar um único vetor de elementos de um determinado tipo, neste caso *double*. A sintaxe geral de declaração para um único vetor é a seguinte:

```
tipo [ ] nome;
```

Nela, os caracteres *[e]* são a indicação de que estão sendo declarado *arrays*. O tipo indica a natureza da informação que estará contida nos *arrays*. Qualquer tipo válido pode ser usado. No primeiro exemplo, *notas* é o nome de um único vetor de números reais.

A sintaxe da segunda linha é utilizada para declarar vários *arrays* distintos de elementos de um mesmo tipo:

```
tipo [ ] nome1, nome2,..., nomen;
```

Nessa sintaxe, *nome1*, *nome2*, etc. são os nomes dos *arrays* que estão sendo criados. Pode haver quantos nomes forem necessários, desde que eles sejam separados por vírgulas. No exemplo mostrado anteriormente, são criados dois vetores de números inteiros: um chamado *idade* e outro chamado *faltas*.

Uma operação fundamental ao lidar com *arrays* é a **alocação**. No momento em que um *array* é declarado, são informados o nome pelo qual ele será referenciado e o tipo dos seus elementos, mas nada é informado a respeito do seu tamanho. Isto faz com que a declaração não crie de fato um *array*, e sim uma referência para um *array*, ou seja, um local que armazena um endereço onde um *array* está ou será posicionado. A tarefa de definir o tamanho de um *array* é feita separadamente durante a alocação. Isso permite que esses tamanhos sejam definidos após a sua declaração.

Alocação

Tarefa de solicitar memória ao sistema operacional, feita durante a execução de programas.

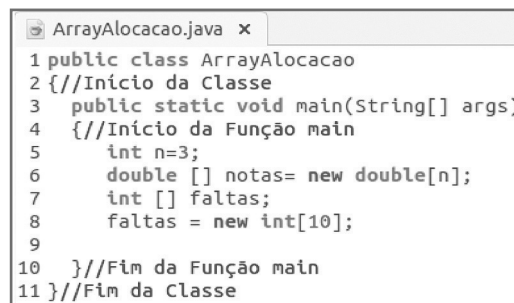


Em Java, todos os objetos são declarados através do uso de referências. Como os *arrays* são objetos em Java, eles também utilizam este mecanismo. Você verá mais detalhes sobre este assunto nas próximas aulas.

O tamanho de um *array* é sempre um número inteiro, independentemente do tipo dos elementos dos *arrays*. Esse tamanho pode ser definido por constantes, variáveis e expressões matemáticas, desde que o resultado seja um número inteiro. Afinal, não faz sentido um conjunto ter “meios elementos”.

A alocação de vetores é feita utilizando-se um operador de atribuição seguido da palavra reservada *new*, seguida de um tipo, que, por sua vez, é seguido de um tamanho entre colchetes (= *new tipo[tamanho]*).

A **Figura 7.1** mostra exemplos de declaração e alocação de vetores. Nessa figura são declarados dois *arrays*, um chamado *notas* e outro chamado *faltas*. Para *notas*, é atribuído um tamanho determinado pela variável *n*, ou seja, *notas* será um vetor com três elementos, no qual cada elemento é um número real. Para *faltas*, é atribuído um tamanho determinado por uma constante (10). Logo, *faltas* será um *array* de 10 elementos, em que cada elemento é um número inteiro.



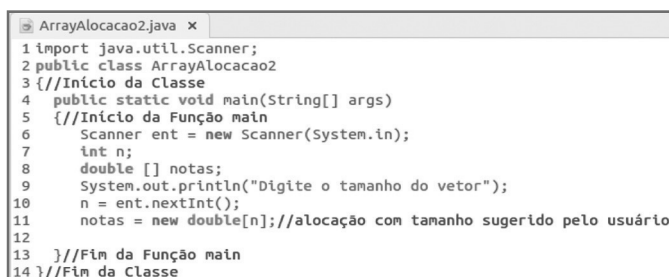
```
ArrayAlocacao.java x
1 public class ArrayAlocacao
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         int n=3;
6         double [] notas= new double[n];
7         int [] faltas;
8         faltas = new int[10];
9     }
10 } //Fim da Função main
11 } //Fim da Classe
```

Figura 7.1: Exemplo de declaração e alocação.

Na sexta linha do algoritmo da **Figura 7.1**, tem-se o *array* *notas* sendo declarado e alocado. Agora note que o mesmo não ocorre com o vetor *faltas*. O *array* *faltas* é declarado na linha 7, mas só é alocado na linha 8. Isso torna possível, por exemplo, fazer um programa que declare um vetor, faça a leitura de uma variável e utilize essa variável como tamanho de um vetor.

A **Figura 7.2** mostra um exemplo dessa possibilidade. Nessa figura, o vetor *notas* é declarado, mas não alocado. Em seguida, é criada a variável *n* e é feita uma interação com o usuário para que este informe

o valor de n . Logo depois, o valor de n é utilizado como tamanho para o vetor *notas*. Logo, se o usuário insere 4 como valor para n , *notas* terá 4 posições.



```

1 import java.util.Scanner;
2 public class ArrayAlocacao2
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         int n;
8         double [] notas;
9         System.out.println("Digite o tamanho do vetor");
10        n = ent.nextInt();
11        notas = new double[n]; //alocação com tamanho sugerido pelo usuário
12    }
13 } //Fim da Função main
14 } //Fim da Classe
  
```

Figura 7.2: Exemplo de alocação com tamanho informado pelo usuário.

Um fato importante a ser mencionado é que os *arrays* não podem ter seu tamanho alterado. Uma vez que se determina que o *array* *faltas* terá 10 posições, não é possível alterar seu tamanho para 11 ou 8. O tamanho de *notas* será sempre 10, embora não seja obrigatório usar todas as posições do *array*. Isso significa que podemos alocar um vetor de 15 posições e utilizar somente 8, por exemplo. Neste contexto, também é importante dizer que é possível fazer duas alocações e atribuí-las à mesma referência, como segue:

```

tint [] v;
v= new int[5];
v=new int[3];
  
```

Contudo, é preciso entender bem o que ocorre neste caso. Na primeira linha, é declarada uma referência para um *array*, chamada v . Na segunda linha, é alocado um vetor de cinco posições e este vetor é atribuído a v . Na terceira linha, é alocado um vetor de três posições e este vetor é atribuído a v . Note que existem duas alocações diferentes. Neste caso no momento da atribuição da linha 3, a referência v deixa de “apontar” para o vetor de cinco posições e passa a “apontar” para o novo vetor de três posições. Portanto, não houve uma mudança no tamanho do vetor referenciado por v , como uma leitura rápida pode sugerir. O que de fato ocorre é que foram alocadas cinco posições e, em seguida, mais três posições. Em muitas linguagens, é preciso **desalocar** as posições alocadas. Em Java, existe um mecanismo que cuida automaticamente da desalocação para o programador, chamado de *Garbage Collector*.

Isto faz com que Java gerencie a desalocação de memória automaticamente, fazendo com que o programador não se preocupe com este passo. Neste exemplo, quando *v* deixa de referenciar o vetor de cinco posições, o número de referências para este vetor passa a ser zero, o que faz com que este vetor seja marcado para que o *Garbage collector* o desaloque posteriormente.

Uma vez que foi definido como declarar vetores, deve-se definir o mecanismo de acesso aos elementos. Java utiliza a seguinte sintaxe para este fim:

```
nome[posição]
```

Nesta sintaxe, *nome* é o identificador do *array* a ser acessado (a referência que “aponta” para ele), e *posição* indica o índice do elemento desejado. É importante mencionar que as posições começam em 0 (zero) e vão até o tamanho do vetor, subtraído de 1. Dessa forma, se temos um *array* de 10 posições, a primeira posição é acessada através do índice 0, a segunda pelo índice 1 e assim sucessivamente até a décima posição, que é acessada através do índice 9. Por exemplo, um *array* de 5 elementos inteiros com as primeiras potências de 2. Nesse caso, como pode ser visto na **Tabela 7.1**, sabendo que o nome do vetor é *potencias2*, pode-se, por exemplo, imprimir o valor da quarta posição, referenciada pelo índice 3. O resultado será a impressão do número 8, que é o valor contido na quarta posição do vetor *potencias2*.

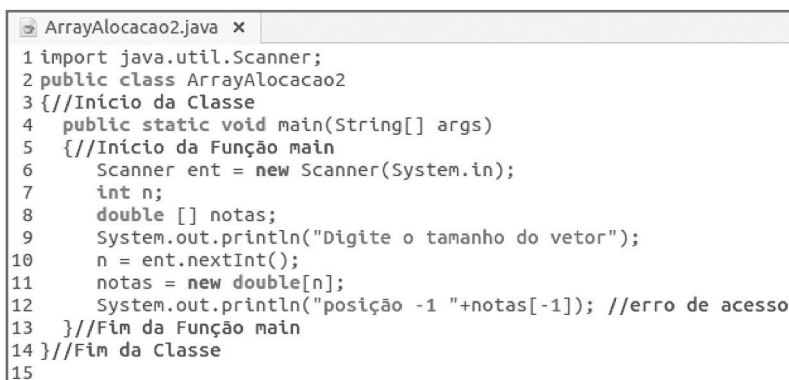
Tabela 7.1: Exemplo de elementos de um vetor e suas posições.

Valores das potências de 2	1	2	4	8	16
Posição	0	1	2	3	4

É importante mencionar que acessar um *array* fora da sua área de indexação é um erro de lógica. Assim, se a indexação de um *array* é de 0 até 4, não podemos acessar as posições anteriores ao 0, nem posteriores ao 4. A **Figura 7.3** mostra o tipo de erro que ocorre quando tentamos acessar um *array* fora da sua indexação.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
at ArrayAlocacao2.main(ArrayAlocacao2.java:12)
```

Figura 7.3: Erro exibido ao acessar uma posição fora de um vetor.



```
1 import java.util.Scanner;
2 public class ArrayAlocacao2
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         int n;
8         double [] notas;
9         System.out.println("Digite o tamanho do vetor");
10        n = ent.nextInt();
11        notas = new double[n];
12        System.out.println("posição -1 "+notas[-1]); //erro de acesso
13    } //Fim da Função main
14 } //Fim da Classe
15
```

Figura 7.4: Programa com acesso à posição inexistente de um vetor.

A **Figura 7.4** mostra o programa que gerou o erro da **Figura 7.3**. Como pode ser observado, este programa nada mais é do que o programa da **Figura 7.2** com uma tentativa de acesso ao índice (posição) -1 do *array*. Note que, na **Figura 7.3**, é exibido o nome do arquivo e a linha em que ocorreu o erro (*ArrayAlocacao2.java:12*). Também é mostrada, na **Figura 7.3**, a posição errada que tentamos acessar (-1). Estas informações são frequentemente úteis na correção de programas, portanto sempre preste atenção nelas ao corrigir os erros dos seus programas.

A **Figura 7.5** mostra um exemplo de uso da sintaxe de acesso. Nesse programa, é criado um vetor de 10 posições, referenciado por *notas*. Em seguida, a cada posição, é atribuído um número múltiplo de 10, de modo que a posição de índice 0 contenha o valor 10, a posição de índice 1 contenha o valor 20 e assim sucessivamente, até a posição de índice 9, que contém o valor 100. Note que, apesar de os valores dos índices dos vetores serem numerados de 0 a 9, qualquer valor pode ser atribuído a um componente do vetor. Neste exemplo, foram usados números múltiplos de 10, mas poderiam ser múltiplos de 15 ou qualquer outro valor válido. Por último, o programa imprime cada um dos elementos dos vetores a partir de seus índices.

```

ArrayExemplo.java x
1 import java.util.Scanner;
2 public class ArrayExemplo
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         double [] notas = new double[10];
8
9         for(int i=0;i<10;i++)//repetição para preencher o vetor
10        {
11            notas[i] = (i+1)*10;
12        }
13
14        for(int i=0;i<10;i++)//repetição para imprimir o vetor
15        {
16            System.out.println(notas[i]);
17        }
18    } //Fim da Função main
19 } //Fim da Classe

```

Figura 7.5: Exemplo de acesso aos elementos de um vetor.

Como os elementos contidos no vetor são os números de 10 a 100, o algoritmo da **Figura 7.5** imprime a seguinte saída:

```

10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.0

```

Atividade 1

Atende aos objetivos 1 e 2

Uma empresa produz cinco produtos e quer saber qual o percentual de vendas de cada produto. Faça um programa que receba a quantidade vendida de cada produto, calcule o total de vendas e o percentual de cada produto neste total.

This image shows a single sheet of white paper with horizontal blue or grey ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.**Resposta comentada**

Neste exemplo, são utilizadas estruturas de repetição juntamente com vetores para atingir os objetivos. Para tanto, a primeira coisa a ser feita é armazenar a quantidade vendida de cada produto. Para isso, pode ser utilizado um vetor de números inteiros. Uma vez feito isso, o total de vendas pode ser calculado a partir da soma dos elementos do vetor. Os percentuais podem ser obtidos através de uma regra de três.

Uma possível solução para o problema, utilizando as estratégias mencionadas, é mostrada a seguir. Nessa solução, a primeira estrutura de repetição trata da leitura dos valores vendidos e do cálculo do total de vendas. Os percentuais são calculados e exibidos na segunda estrutura de repetição. Note que, durante a regra de três, é necessário fazer uma multiplicação por uma constante real, devido ao fato da variável *soma* e os valores do vetor *vendas* serem inteiros.

```
ArrayPercentualVendas.java x
1 import java.util.Scanner;
2 public class ArrayPercentualVendas
3 { //Início da classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7
8         int [] vendas = new int[5];
9         for(int i=0;i<5;i++)//repetição para leitura de dados
10        {
11            System.out.println("Digite as vendas para o produto "+(i+1));
12            vendas[i]= ent.nextInt();
13        }
14        int soma=0;
15        for(int i=0;i<5;i++)//repetição para leitura soma das vendas
16        {
17            soma+=vendas[i];
18        }
19        for(int i=0;i<5;i++)//repetição para calculo dos percentuais
20        {
21            double perc = 100.0*vendas[i]/soma;
22            System.out.println("percentual do produto "+(i+1)+" é "+perc+"%");
23        }
24    } //Fim da Função main
25 } //Fim da Classe
```

Figura 7.6: O total de vendas pode ser calculado a partir da soma dos elementos do vetor.

Matrizes: declaração e acesso

Em muitos casos, tratar armazenamento com uma indexação única não é conveniente. Um exemplo típico são os calendários utilizados no dia a dia. Em suas linhas, o calendário de um dado mês armazena os dias do mês que estão na mesma semana. Nas colunas, estão os dias do mês que caem no mesmo dia da semana. A **Figura 7.7** mostra um exemplo de calendário. Note que, ao inspecionar a última linha da tabela, verificamos todos os dias que pertencem à ultima semana do mês em questão. Ao inspecionarmos a sexta coluna, encontramos todos os dias que caem em um sábado.

Segunda-Feira	Terça-Feira	Quarta-Feira	Quinta-Feira	Sexta-Feira	Sábado	Domingo
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Figura 7.7: Calendário, exemplo de matriz.

Nesse caso, a representação por tabela ou matriz é mais conveniente do que uma representação por vetor. Matrizes também são muito naturais em diversos problemas da matemática e da engenharia e, por isso, seu estudo é objeto de diversas áreas. Para exemplificar, matrizes são utilizadas na resolução de sistemas de equações, no processamento de imagens, em simulações, geoprocessamentos e em muitas outras situações. Devido a isso, é de fundamental importância saber como construir programas capazes de manipular matrizes, uma vez que muitas das tarefas mencionadas, embora possam ser feitas manualmente em alguns casos, na maioria deles, inclusive nos mais complexos, é necessário o uso de computadores.

Matrizes, ou *arrays* multidimensionais, também são um conjunto identificável de posições de memória, em que se podem armazenar informações de um tipo específico. Elas podem ser vistas como um conjunto de elementos do mesmo tipo, no qual é possível identificar, acessar e alterar cada elemento. A diferença entre uma matriz e um *array* (ou vetor) é o número de indexações necessárias para se fazer o acesso. Como o *array* só possui uma dimensão, apenas um índice é necessário. Já com as matrizes, é preciso um índice para cada dimensão. Ou seja, para acessar as informações de uma matriz de duas dimensões, são necessários dois índices; para uma matriz de três dimensões são necessários três índices, e assim sucessivamente.

A declaração de referências para matrizes segue a seguinte sintaxe:

```
tipo [ ] [ ] nome;
tipo [ ] [ ] [ ] nome1, nome2,..., nomen;
```

A primeira linha é utilizada para declarar uma única matriz de duas dimensões. A segunda é utilizada para declarar várias matrizes de três dimensões. Nessa sintaxe, nome1, nome2, etc., são os nomes das referências para matrizes que serão criados e pode haver quantos nomes forem necessários, desde que eles sejam separados por vírgulas. Cada par de colchetes (“[” e “]”) indica uma dimensão. Logo, para matrizes de três dimensões, são necessários três pares de colchetes e, assim, não há limite para o número de dimensões. O tipo indica a natureza da informação que estará contida nas matrizes; qualquer tipo válido pode ser usado.

Para exemplificar o uso de matrizes, considere a seguinte situação: uma empresa tem quatro vendedores, que vendem cinco produtos. No final do dia, cada vendedor entrega uma nota de cada tipo de produto diferente vendido. Cada nota contém:

1. número do vendedor;
2. número do produto e
3. valor total vendido desse produto em reais.

Faça um algoritmo que leia as notas de cada produto para cada vendedor e calcule a comissão de cada um deles. A comissão é calculada pela seguinte equação:

$$\text{Comissão} = 10\% * (\text{totalProd1}) + 20\% * (\text{totalProd2}) + 10\% * (\text{totalProd3}) + 20\% * (\text{totalProd4}) + 10\% * (\text{totalProd5}).$$

Essa situação é típica para o uso de matrizes. Note que a situação descreve que cada vendedor entrega uma nota para cada produto, mas nada é mencionado a respeito da ordem de processamento das notas. Se fosse garantido que todas as notas de um mesmo vendedor fossem processadas em conjunto, seria possível conseguir calcular a comissão de cada um facilmente, mas isso não necessariamente acontece. O funcionário do setor financeiro pode processar as notas em ordem aleatória ou, por exemplo, pode processar as notas agrupando-as por produto. Nesse sentido, devem-se armazenar os dados em uma matriz que contém os dados de todos os vendedores para cada produto. Só depois de ter armazenado todos os dados é que é possível calcular a comissão de cada funcionário. A solução desse problema é descrita detalhadamente mais adiante. Por enquanto, vamos aprender a declarar corretamente as matrizes e utilizá-las corretamente em algoritmos.

A **Figura 7.8** mostra um exemplo de declaração, preenchimento e impressão de matriz. Uma matriz de duas dimensões com três linhas e três colunas é criada na linha 6. As linhas de 8 a 14 tratam do preenchimento da matriz. Note que, assim como nos vetores, os índices das matrizes começam em zero. Desse modo, os índices das linhas e das colunas, neste exemplo, vão de 0 a 2. As linhas de 16 a 22 tratam da impressão dos elementos da matriz.

```

MatrizExemplo.java x
1 public class MatrizExemplo
2 //Inicio da Classe
3 public static void main(String[] args)
4 //Inicio da Função main
5
6     int [][] num = new int[3][3];
7
8     for(int i=0;i<3;i++)//repetições para preencher a matriz
9     {
10         for(int j=0;j<3;j++)
11         {
12             num[i][j]= i*3+j+1;
13         }
14     }
15
16     for(int i=0;i<3;i++)//repetições para imprimir a matriz
17     {
18         for(int j=0;j<3;j++)
19         {
20             System.out.println(num[i][j]);
21         }
22     }
23 }//Fim da Função main
24 }//Fim da Classe

```

Figura 7.8: Exemplo de declaração, preenchimento e impressão de uma matriz de números reais com duas dimensões.

Note que, o programa da **Figuras 7.8** imprime os elementos como se fossem uma única coluna. Isso ocorre porque cada comando de impressão *System.out.println* sempre utiliza uma linha. Na repetição das linhas 18 a 21, são impressos, para um dado *i*, todos os *j* possíveis, ou seja, três valores são impressos para cada *i*, tomando, então, três linhas. Como os múltiplos valores de *i* são tratados pela repetição que começa na linha 16, e são tratados três possíveis valores para *i*, temos que a repetição das linhas de 18 a 21 é executada três vezes. Logo, são impressos $3 \times 3 = 9$ elementos no total. A impressão do programa da **Figura 7.8** é exibida do seguinte modo:

```

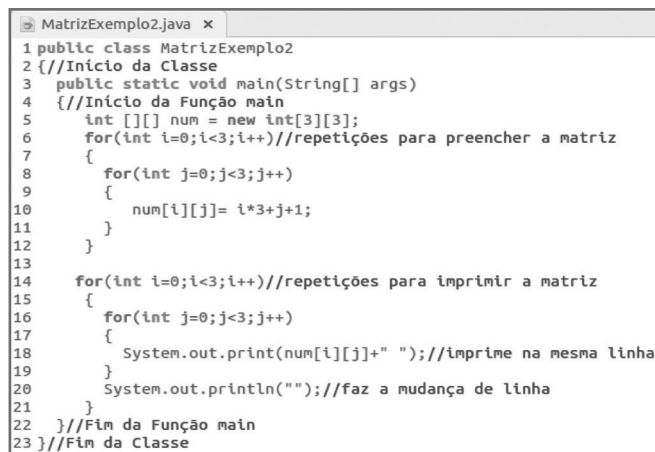
1
2
3
4
5
6
7
8
9

```

Observe que esse não é o modo usual utilizado para exibir uma matriz de duas dimensões. Fica difícil diferenciar uma matriz impressa dessa maneira de um vetor com nove posições. Para podermos visualizar

as nuances de uma matriz, o ideal é representá-la com linhas e colunas, como é feito tradicionalmente na matemática. A **Figuras 7.9** mostra como se faz isso. Note que a parte inicial é igual ao programa da **Figuras 7.8**. A diferença está nas repetições que tratam da impressão.

A **Figura 7.9** utiliza o comando `System.out.print` ao invés do `System.out.println`. Isso faz com que os valores sejam impressos na mesma linha. O ponto crítico é mudar de linha na hora certa. Observe que, na linha 20, é feita uma impressão com o comando `println`. Este comando é que faz a mudança de linha. Isso faz com que, na primeira execução, seja impresso o elemento `num[0][0]`, seguido de um espaço, ou seja “1 ”. Depois, o valor contido em `num[0][1]` é impresso seguido de outro espaço, ou seja, “1 2 ”. O mesmo ocorre para `num[0][2]`, fazendo que seja impresso “1 2 3 ”. Note que, nesse ponto, a repetição em `j` termina e o comando `println` é invocado, fazendo a mudança de linha na tela. Quando `i` muda seu valor para 1, a repetição em `j` será executada novamente, fazendo com que os elementos da matriz na linha 1 sejam impressos na mesma linha.



```

1 public class MatrizExemplo2
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         int [][] num = new int[3][3];
6         for(int i=0; i<3; i++) //repetições para preencher a matriz
7         {
8             for(int j=0; j<3; j++)
9             {
10                 num[i][j] = i*3+j+1;
11             }
12         }
13         for(int i=0; i<3; i++) //repetições para imprimir a matriz
14         {
15             for(int j=0; j<3; j++)
16             {
17                 System.out.print(num[i][j]+" "); //imprime na mesma linha
18             }
19             System.out.println(""); //faz a mudança de linha
20         }
21     }
22 } //Fim da Função main
23 } //Fim da Classe

```

Figura 7.9: Impressão de matriz no formato linha x coluna.

A saída do programa da **Figura 7.9** é mostrada a seguir. Note que este é um modo bem mais inteligível para se visualizar uma matriz.

```

1 2 3
4 5 6
7 8 9

```


Assim como em *arrays*, acessar uma posição fora da área de indexação em matrizes também é um erro de lógica. Isso é válido para qualquer dimensão, ou seja, não adianta acessar corretamente as linhas de uma matriz de duas dimensões se, ao acessar as colunas, utiliza-se um índice -1, por exemplo. Mesmo que apenas uma das várias dimensões de uma matriz seja acessada erroneamente, o programa estará errado.

Atividade 2

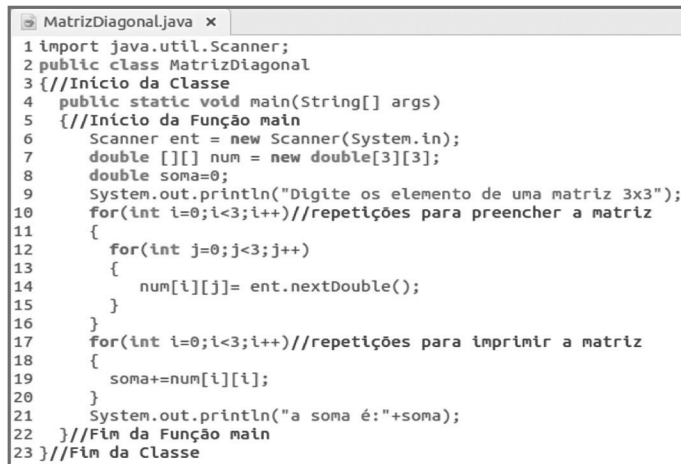
Atende aos objetivos 1 e 2

Faça um programa que calcule a soma dos elementos da diagonal principal de uma matriz 3 x 3 de números reais. Os números da matriz devem ser informados pelo usuário.

[illegible]**Resposta comentada**

Os primeiros passos para resolver essa questão são declarar as variáveis necessárias e fazer a leitura dos elementos da matriz. Como estes são números reais, a soma dos elementos da diagonal principal também deve ser declarada como real. O próximo passo é fazer a soma. Note que, na diagonal principal de uma matriz, o índice da linha é sempre igual ao índice da coluna. Assim, para controlar a mudança entre os elementos, apenas uma estrutura de repetição é necessária. A seguir, é mostrada uma possível solução para o problema. Nela, a soma é feita pela repetição das linhas 17-20, onde à variável soma é adicionado o valor da matriz na posição (i,i). Ou seja, na primeira iteração, será adicionada à

soma a posição (0,0), depois a posição (1,1) e assim sucessivamente. No final, o resultado da soma dos elementos da diagonal principal é exibido.



```

1 import java.util.Scanner;
2 public class MatrizDiagonal
3 { //Inicio da Classe
4     public static void main(String[] args)
5     { //Inicio da Função main
6         Scanner ent = new Scanner(System.in);
7         double [][] num = new double[3][3];
8         double soma=0;
9         System.out.println("Digite os elemento de uma matriz 3x3");
10        for(int i=0;i<3;i++)//repetições para preencher a matriz
11        {
12            for(int j=0;j<3;j++)
13            {
14                num[i][j]= ent.nextDouble();
15            }
16        }
17        for(int i=0;i<3;i++)//repetições para imprimir a matriz
18        {
19            soma+=num[i][i];
20        }
21        System.out.println("a soma é:"+soma);
22    } //Fim da Função main
23 } //Fim da Classe
  
```

Figura 7.10: Aqui a soma é feita pela repetição das linhas 17-20, onde a variável soma é adicionado o valor da matriz na posição (i,i).

Como pôde ser observado ao longo da aula, o armazenamento de dados em *arrays* possibilita compor algoritmos para problemas mais complexos do que os resolvidos até então. Além disso, os *arrays* têm um papel importante na simplificação de certos algoritmos e na criação de estruturas de dados mais complexas, como *heaps*, tabelas *Hash* e árvores binárias, que você estudará em outras disciplinas. Neste sentido, o uso de matrizes e vetores é fundamental para lidar com alguns problemas. Elas podem ser utilizadas tanto para guardar informações simples, como os dias em um calendário, como resultados de cálculos de modelos matemáticos sofisticados. Em outras disciplinas, como Álgebra Linear e Métodos Numéricos, você verá a aplicação de matrizes em soluções de problemas matemáticos e de engenharia, portanto o estudo dos algoritmos e programas relacionados à criação e manipulação de matrizes lhe será muito útil ao longo do seu curso.

Atividade final

Atende aos objetivos 1 e 2

1. Faça um programa que leia e armazene em um vetor um conjunto de números. O programa deve imprimir os números pares e ímpares separadamente sendo que, primeiro, devem ser impressos os pares e depois os ímpares. O tamanho do conjunto deve ser informado pelo usuário.

2. Faça um programa que calcule e imprima a soma dos elementos de cada coluna de uma matriz de números inteiros.

3. Faça um programa que receba, uma a uma, as letras de uma palavra e determine se esta palavra é um palíndromo. Obs.: Um palíndromo é uma palavra que, quando escrita de trás para frente, permanece igual. Exemplos: ovo, radar.

[illegible]

4.

a) Uma empresa tem quatro vendedores, que vendem cinco produtos. No final do dia, cada vendedor entrega uma nota de cada tipo de produto diferente. Cada nota contém: 1. número do vendedor; 2. número do produto e 3. valor total vendido deste produto em reais.

Considere que, mesmo que não tenha vendido nada, o vendedor deve preencher uma nota com o valor total vendido igual a zero. Faça um programa que leia as notas de cada produto para cada vendedor e calcule a comissão de cada um deles. A comissão é calculada pela seguinte equação:

$$\text{Comissão} = 10\% * (\text{totalProd1}) + 20\% * (\text{totalProd2}) + 10\% * (\text{totalProd3}) + 20\% * (\text{totalProd4}) + 10\% * (\text{totalProd5}).$$

[illegible]

b) Agora suponha que esta mesma empresa esteja revendo seus procedimentos internos. Ainda são os mesmos quatro vendedores, que vendem os mesmos cinco produtos. Porém, agora, no final do dia, cada vendedor entrega uma nota de cada tipo de produto diferente vendido. Cada nota contém: 1. número do vendedor; 2. número do produto; 3. dia do mês em que a venda ocorreu e 4. valor total vendido deste produto em reais nesse dia.

O vendedor só é obrigado a preencher a nota caso tenha vendido o produto. No final do mês, o gerente da loja processa as notas de todos os funcionários para calcular a remuneração detalhada de cada um deles. Faça um programa que leia as informações de cada uma das notas, calcule e mostre a comissão diária de cada vendedor para cada dia do mês. Suponha que o número total de notas deve ser fornecido pelo gerente. Suponha que o mês sempre tem 30 dias. Para a comissão diária, devem ser considerados os valores de um mesmo dia. A comissão diária é calculada pela seguinte equação:

$$\text{Comissão} = 10\% * (\text{totalProd1}) + 20\% * (\text{totalProd2}) + 10\% * (\text{totalProd3}) + 20\% * (\text{totalProd4}) + 10\% * (\text{totalProd5}).$$

[illegible]

5. Faça um programa que calcule e imprima os termos da série de Fibonacci com o auxílio de um vetor. A função que descreve a série de Fibonacci é dada por: $f(1) = 1$, $f(2) = 1$ e $f(n) = f(n-1) + f(n-2)$. O número de termos deve ser informado pelo usuário.

```

ArrayParesImpares.java x
1 import java.util.Scanner;
2 public class ArrayParesImpares
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         int n;
8         System.out.println("Digite o tamanho do vetor");
9         n=ent.nextInt();
10        int [] num = new int[n];
11        for(int i=0;i<n;i++)//repetição para leitura de dados
12        {
13            System.out.println("Digite um número");
14            num[i]= ent.nextInt();
15        }
16        System.out.println("números pares");
17        for(int i=0;i<n;i++)//repetição para impressão dos pares
18        {
19            if(num[i]%2==0)
20                System.out.println(num[i]);
21        }
22        System.out.println("números impares");
23        for(int i=0;i<n;i++)//repetição para impressão dos impares
24        {
25            if(num[i]%2!=0)
26                System.out.println(num[i]);
27        }
28    } //Fim da Função main
29 } //Fim da Classe

```

Figura 7.11: Uma das possíveis soluções para este problema.

2. Existem vários mecanismos para resolver essa questão. Um dos mais sofisticados é utilizar, além da matriz mencionada, um vetor para armazenar a soma de cada uma das colunas. A seguir é apresentada uma possível solução, que usa esta estratégia, para esse problema.

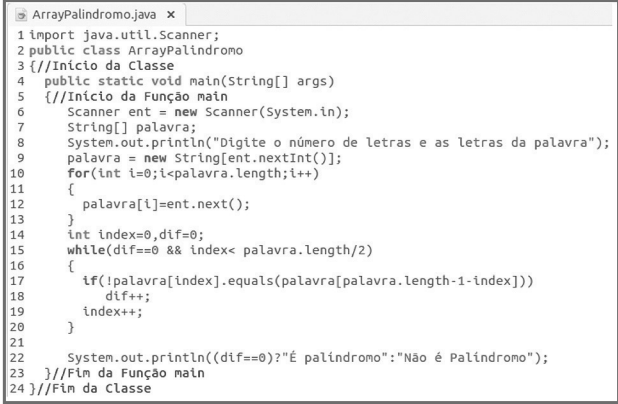
```

MatrizSomaColuna.java x
1 import java.util.Scanner;
2 public class MatrizSomaColuna
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent= new Scanner(System.in);
7         System.out.println("Digite as dimensões da matriz");
8         int lin = ent.nextInt();
9         int col = ent.nextInt();
10        int [][] num = new int[lin] [col];
11        System.out.println("Digite os numeros da matriz");
12        for(int i=0;i<lin;i++)//repetições para preencher a matriz
13        {
14            for(int j=0;j<col;j++)
15            {
16                num[i][j]= ent.nextInt();
17            }
18        }
19        int [] soma=new int[col];
20        for(int j=0;j<col;j++)//calculando as somas das colunas
21        {
22            for(int i=0;i<lin;i++)
23            {
24                soma[j]+= num[i][j];
25            }
26        }
27        System.out.println("as somas das colunas são");
28        for(int j=0;j<col;j++)
29            System.out.print(soma[j]+" ");
30        System.out.println();
31    } //Fim da Função main
32 } //Fim da Classe

```

Figura 7.12: Exemplo de solução que utiliza, além da matriz mencionada, um vetor para armazenar a soma de cada uma das colunas.

3. Existem vários modos de se resolver esta questão. Um deles é o uso explícito de vetores para armazenar cada uma das letras. Essa solução é mostrada a seguir. A estratégia é simples: percorrer o vetor de letras, comparando a primeira com a última, a segunda com a penúltima e assim sucessivamente até a metade do vetor. Caso alguma destas comparações indique letras diferentes, a variável de marcação *dif* terá seu valor alterado, indicando que existe uma diferença. Ao final da comparação, se *dif* mantiver o valor zero, isso significa que nenhuma diferença foi encontrada, logo, a palavra é um palíndromo.



```

1 import java.util.Scanner;
2 public class ArrayPalindromo
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         String[] palavra;
8         System.out.println("Digite o número de letras e as letras da palavra");
9         palavra = new String[ent.nextInt()];
10        for(int i=0; i<palavra.length; i++)
11        {
12            palavra[i]=ent.next();
13        }
14        int index=0, dif=0;
15        while(dif==0 && index< palavra.length/2)
16        {
17            if(!palavra[index].equals(palavra[palavra.length-1-index]))
18                dif++;
19            index++;
20        }
21        System.out.println((dif==0)? "É palindromo": "Não é Palindromo");
22    } //Fim da Função main
23 } //Fim da Classe
  
```

Figura 7.13: Solução que emprega o uso explícito de vetores para armazenar cada uma das letras.

Neste exercício, é utilizado o membro *length*, do *array* palavras. Em várias linguagens, quando temos múltiplos *arrays* com tamanhos diferentes, existe a necessidade de criar múltiplas variáveis para armazenar estes tamanhos. Em Java, cada *array* sabe seu próprio tamanho e, para acessar esta informação, só é preciso fazer um acesso ao membro *length* do *array* em questão.

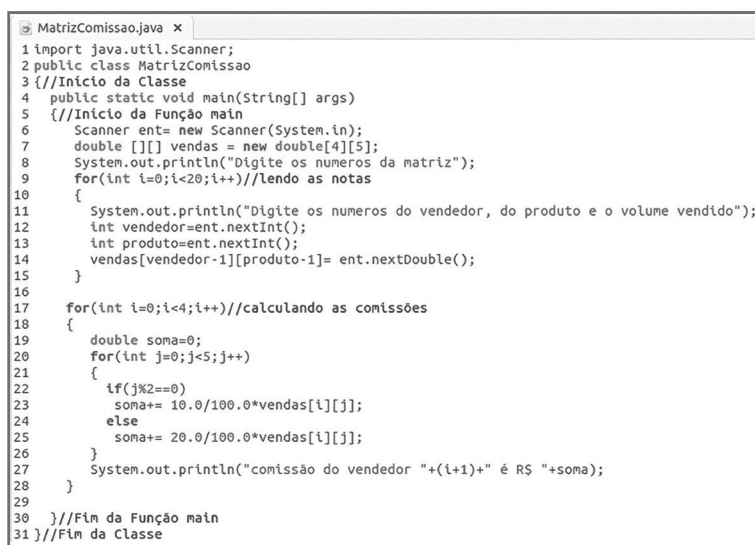
Um modo mais racional de resolver esta questão é aplicar a ela a mesma lógica apresentada no parágrafo anterior, mas, ao invés de utilizar explicitamente um vetor, utilizar os membros da classe *String* (como o *charAt* por exemplo), apresentados na Aula 2. Tais membros simplificam o uso de variáveis do tipo texto, uma vez que fica claro que elas são, de fato, *arrays*.

4.

a) Para resolver essa tarefa, deve-se utilizar uma matriz bidimensional e estabelecer o que as linhas e o que as colunas representarão. As linhas

podem ser usadas para representar vendedores, e as colunas, produtos, ou vice-versa. Uma possível solução, utilizando as linhas como vendedores é apresentada a seguir. Nesta solução, é criada uma matriz de quatro linhas por cinco colunas (uma linha para cada vendedor e uma coluna para cada produto). Desse modo, para calcular a comissão de um vendedor, deve-se percorrer a linha correspondente a ele. Cada coluna dessa linha representa o montante vendido, em reais, para um determinado produto.

Note que a equação da comissão apresenta 10% para o total dos produtos ímpares e 20% para os produtos pares. Contudo, a equação apresenta produtos que começam em 1 e vão até 5. A matriz que está sendo utilizada possui indexação de colunas começando em 0 e indo até 4. Nesse sentido, é preciso fazer uma pequena adaptação, visto que os dados referentes ao produto 1 estarão armazenados na coluna 0, os dados do produto 2 estarão armazenados na coluna 1, e assim sucessivamente, até os dados do produto 5, que estarão armazenados na coluna 4. Ajustes similares também devem ser feitos para que os dados do vendedor 1 estejam armazenados na linha 0 da matriz, e os dados do vendedor 4 sejam armazenados na linha 3. Note que a solução permite que os dados sejam inseridos fora da ordem, e só depois de inseridos os dados de todas as 20 notas (5 produtos x 4 vendedores) é que o programa começa a calcular as comissões.



```

1 import java.util.Scanner;
2 public class MatrizComissao
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent= new Scanner(System.in);
7         double [][] vendas = new double[4][5];
8         System.out.println("Digite os numeros da matriz");
9         for(int i=0;i<20;i++)//lendo as notas
10        {
11            System.out.println("Digite os numeros do vendedor, do produto e o volume vendido");
12            int vendedor=ent.nextInt();
13            int produto=ent.nextInt();
14            vendas[vendedor-1][produto-1]= ent.nextDouble();
15        }
16
17        for(int i=0;i<4;i++)//calculando as comissões
18        {
19            double soma=0;
20            for(int j=0;j<5;j++)
21            {
22                if(j%2==0)
23                    soma+= 10.0/100.0*vendas[i][j];
24                else
25                    soma+= 20.0/100.0*vendas[i][j];
26            }
27            System.out.println("comissão do vendedor "+(i+1)+" é R$ "+soma);
28        }
29    } //Fim da Função main
30 } //Fim da Classe
  
```

Figura 7.14: Possível solução que utiliza as linhas como vendedores.

b) Esse problema requer o uso de uma matriz de três dimensões. Uma para representar os vendedores, outra para representar os produtos e uma terceira para representar os dias. Uma das informações que são passadas pelo enunciado é a de que somente as vendas são reportadas; portanto, o vendedor não é obrigado a preencher notas de um produto quando ele não fez vendas daquele produto. Isso indica que pode não haver notas para todos os funcionários/dias/produtos. Outra informação dada é a de que o número de notas deve ser informado pelo gerente e essa informação facilitará o processo de leitura dos dados.

Uma vez tendo os dados colocados na matriz de maneira correta, deve-se calcular para cada vendedor a comissão de cada dia. Para isso, é necessário fixar um vendedor e um dia, e percorrer todos os produtos vendidos naquele dia para calcular sua comissão. Ao término do cálculo, deve-se imprimir a comissão do dia e repetir essa tarefa para todos os dias e para todos os vendedores. A seguir está uma possível solução que utiliza esta estratégia.

```

MatrizComissao2.java x
1 import java.util.Scanner;
2 public class MatrizComissao2
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent= new Scanner(System.in);
7         double [][][] vendas = new double[4][5][30];
8         System.out.println("Digite o numero de notas");
9         int numNotas=ent.nextInt();
10        for(int i=0;i<numNotas;i++)//lendo as notas
11        {
12            System.out.println("Digite os numeros do vendedor, do produto, o dia e o volume vendido");
13            int vendedor=ent.nextInt();
14            int produto=ent.nextInt();
15            int dia=ent.nextInt();
16            vendas[vendedor-1][produto-1][dia-1]= ent.nextDouble();
17        }
18    }
19    for(int t=0;t<vendas[0][0].length;t++)//repetição nos dias
20    {
21        System.out.println("dia "+(t+1));
22        for(int i=0;i<vendas.length;i++)//repetição nos vendedores
23        {
24            double soma=0;
25            for(int j=0;j<vendas[0].length;j++)//repetição nos produtos
26            {
27                if(j%2==0)
28                    soma+= 10.0/100.0*vendas[i][j][t];
29                else
30                    soma+= 20.0/100.0*vendas[i][j][t];
31            }
32            System.out.println("comissão do vendedor "+(i+1)+" é R$ "+soma);
33        } //fim da repetição nos vendedores
34    } //fim da repetição nos dias
35    }
36 } //fim da Função main
37 } //fim da Classe

```

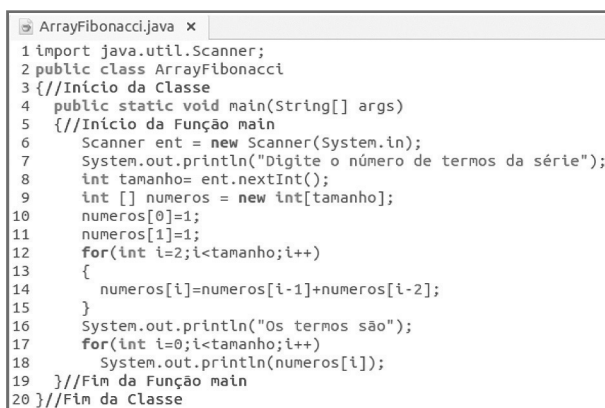
Figura 7.15: Possível solução que fixa um vendedor e um dia, para daí percorrer todos os produtos vendidos naquele dia e calcular sua comissão.

Note que foram acessados os membros *length* de partes da matriz (*vendas[0].length*, por exemplo). Isto permite ter acesso ao tamanho de cada dimensão de uma matriz. Se, por exemplo, desejamos saber quantas colunas há na terceira linha de uma matriz chamada *nomes*, basta aces-

sarmos o membro *length* da linha em questão, ficando *nomes[2].length* neste caso.

5. Esta atividade é apenas ilustrativa, uma vez que já foi visto na disciplina de Computação I que a série de Fibonacci pode ser calculada sem o uso de vetores. A seguir está uma possível solução para a questão. Primeiramente é criado um vetor para conter os termos da série. A variável *tamanho* representa a quantidade de termos da série que devemos calcular, logo, o vetor *numeros* deve ter um número de posições adequado ao valor de *tamanho*. Os dois primeiros termos da série são conhecidos da definição matemática ($f(n)$ resulta em 1, se $n = 1$ ou $n = 2$; $f(n)$ resulta em $f(n-1) + f(n-2)$, caso contrário).

Assim, as posições de índice 0 e 1 (primeira e segunda posições) são inicializadas com 1. Desse modo, é preciso calcular os valores da série do terceiro termo em diante, por isso a estrutura de repetição *Para* faz a inicialização da variável *i* em 2 (terceira posição). Nesta primeira estrutura de repetição, o termo atual da série (*i*) é calculado em função dos dois termos anteriores (*i - 1* e *i - 2*). A segunda estrutura de repetição é utilizada para imprimir os termos calculados.



```

ArrayFibonacci.java x
1 import java.util.Scanner;
2 public class ArrayFibonacci
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         System.out.println("Digite o número de termos da série");
8         int tamanho= ent.nextInt();
9         int [] numeros = new int[tamanho];
10        numeros[0]=1;
11        numeros[1]=1;
12        for(int i=2;i<tamanho;i++)
13        {
14            numeros[i]=numeros[i-1]+numeros[i-2];
15        }
16        System.out.println("Os termos são");
17        for(int i=0;i<tamanho;i++)
18            System.out.println(numeros[i]);
19    } //Fim da Função main
20 } //Fim da Classe
  
```

Figura 7.16: Possível solução para a questão.

6. A intercalação, neste caso, consiste em colocar as posições de índice iguais como vizinhas no vetor resultante. Uma possível solução é dada a seguir. Nessa solução são criados três vetores: *vet1* e *vet2* são os vetores de entrada e *vet3* é o vetor em que será armazenada a intercalação de *vet1* e *vet2*. Note que é possível determinar as posições dos vetores *vet1* e *vet2* a serem acessadas a partir das posições de *vet3*.

```

ArrayIntercalacao.java x
1 import java.util.Scanner;
2 public class ArrayIntercalacao
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         int[] vet1, vet2, vet3;
8         vet1 = new int[5];
9         vet2 = new int[5];
10        vet3 = new int[10];
11        System.out.println("Digite os termos do vetor 1");
12        for(int i=0; i<5; i++)
13        {
14            vet1[i] = ent.nextInt();
15        }
16        System.out.println("Digite os termos do vetor 2");
17        for(int i=0; i<5; i++)
18        {
19            vet2[i] = ent.nextInt();
20        }
21        for(int i=0; i<10; i++) //calculando a intercalação
22        {
23            if(i%2==0)
24                vet3[i] = vet1[i/2];
25            else
26                vet3[i] = vet2[i/2];
27        }
28        System.out.println("A intercalação é :");
29        for(int i=0; i<10; i++)
30        {
31            System.out.println(vet3[i]);
32        }
33    } //Fim da Função main
34 } //Fim da Classe

```

Figura 7.17: A intercalação, neste caso, consiste em colocar as posições de índice iguais como vizinhas no vetor resultante.

7. A primeira coisa a se observar é a diferença entre os conceitos de vetor geométrico (às vezes chamado de vetor algébrico ou simplesmente vetor) e vetor computacional (*arrays*). Em português, eles têm o mesmo nome (vetor). Devido a isso, para esta questão, faremos uma distinção de termos. Quando formos tratar dos vetores geométricos, utilizaremos o termo *vetor*. Quando formos tratar dos vetores computacionais, utilizaremos o termo *array*. Um vetor é uma entidade que possui associados um valor, uma direção e um sentido. Um *array* é só um conjunto de valores (mais informações sobre vetores em STEINBRUCH; WINTERLE, 1987). Entretanto, uma das formas de se representar um vetor é através de suas coordenadas cartesianas. Para vetores em R^3 , são necessários três coordenadas, uma para o eixo x , uma para o eixo y e uma para o eixo z . Uma possível solução é dada a seguir. Nesta solução, são criados dois *arrays* de três posições: *vet1*, *vet2* (entrada). Ou seja, usaremos um *array* para representar um vetor através do armazenamento de suas coordenadas cartesianas. A primeira observação a ser feita é que o produto escalar entre dois vetores resulta em um escalar, ou seja, um número. As linhas 21-24 cuidam da implementação da equação, que é um somatório dos produtos das coordenadas de mesma posição dos vetores.

```

ArrayIntercalacao.java x
1 import java.util.Scanner;
2 public class ArrayIntercalacao
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         int[] vet1, vet2, vet3;
8         vet1 = new int[5];
9         vet2 = new int[5];
10        vet3 = new int[10];
11        System.out.println("Digite os termos do vetor 1");
12        for(int i=0; i<5; i++)
13        {
14            vet1[i] = ent.nextInt();
15        }
16        System.out.println("Digite os termos do vetor 2");
17        for(int i=0; i<5; i++)
18        {
19            vet2[i] = ent.nextInt();
20        }
21        for(int i=0; i<10; i++) //calculando a intercalação
22        {
23            if(i%2==0)
24                vet3[i] = vet1[i/2];
25            else
26                vet3[i] = vet2[i/2];
27        }
28        System.out.println("A intercalação é :");
29        for(int i=0; i<10; i++)
30        {
31            System.out.println(vet3[i]);
32        }
33    } //Fim da Função main
34 } //Fim da Classe

```

Figura 7.18: Nesta solução, são criados dois de três posições: *vet1*, *vet2* (entrada).

8. O problema da variância já foi detalhado na introdução desta aula. Uma possível solução para ele é mostrada a seguir.

```

ArrayVariancia.java x
1 import java.util.Scanner;
2 public class ArrayVariancia
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         System.out.println("Digite a quantidade de números");
8         double [] num = new double[ent.nextInt()];
9         System.out.println("Digite os números");
10        for(int i=0; i<num.length; i++)
11        {
12            num[i] = ent.nextDouble();
13        }
14        double media=0;
15        for(int i=0; i<num.length; i++)
16        {
17            media += num[i];
18        }
19        media /= num.length;
20        double var=0;
21        for(int i=0; i<num.length; i++)
22        {
23            var += Math.pow((num[i]-media), 2.0);
24        }
25        var /= num.length;
26        System.out.println("variância=" + var);
27    }
28 } //Fim da Função main
29 } //Fim da Classe

```

Figura 7.19: Possível solução para esta questão de cálculo de variância.

9. Nesta questão, os dados são lidos primeiro e depois devem ser ordenados. Desse modo, uma possível solução, que utiliza uma estratégia chamada de ordenação por seleção, é apresentada a seguir. Essa solução percorre o *array*, procurando o elemento de prioridade (neste caso, o menor elemento). Uma vez encontrado o elemento de prioridade, este elemento é inserido na posição adequada. Para garantir a ordenação, é necessário realizar este procedimento para cada posição do vetor.

```

ArrayOrdenacao.java x
1 import java.util.Scanner;
2 public class ArrayOrdenacao
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         System.out.println("Digite a quantidade de números");
8         double [] num = new double[ent.nextInt()];
9         System.out.println("Digite os números");
10        for(int i=0;i<num.length;i++)//leitura
11        {
12            num[i]=ent.nextDouble();
13        }
14        for(int i=0;i<num.length-1;i++)//ordenação
15        {
16            double menor= num[i];
17            int pos = i;
18            for(int j=i+1;j<num.length;j++)//busca elemento de prioridade
19            {
20                if(num[j]<menor)//toda vez que um novo elemento de maior
21                { //prioridade é encontrado substituí-se o antigo
22                    menor=num[j];
23                    pos=j;
24                }
25            }
26            if(pos!=i)//se pos foi alterado, um elemento de maior prioridade
27            { // foi encontrado.
28                double aux= num[i];//trocando o elemento atual com o de
29                num[i]=menor; //prioridade
30                num[pos]=aux;
31            }
32        }
33        System.out.println("Sequencia ordenada");
34        for(int i=0;i<num.length;i++)//impressão
35        {
36            System.out.println(num[i]);
37        }
38    } //Fim da Função main
39 } //Fim da Classe

```

Figura 7.20: Possível solução, que utiliza uma estratégia chamada de ordenação por seleção.

10. Este também é um programa bem extenso, caso todas as tarefas sejam feitas separadamente. A seguir, está uma possível solução do problema. A primeira parte trata da inicialização e da leitura dos dados. A segunda trata da multiplicação das matrizes e a terceira, da impressão da matriz resultante. Para relembrar, a equação para calcular a multiplicação de matrizes é $c_{ij} = \sum_{k=1}^m a_{ik} \cdot b_{kj}$, $\forall i \in \{1, \dots, l\}, \forall j \in \{1, \dots, n\}$, sendo c

a matriz resultante, a e b as duas matrizes a serem multiplicadas, l o número de linhas da matriz a , m o número de colunas da matriz a (e consequentemente o número de linhas de b) e n o número de colunas da matriz b . Note que, para encontrar um elemento da matriz c , é preciso fazer uma série de multiplicações dos elementos da matriz a por elementos da matriz b e, depois, somar os resultados dessas multiplicações. As linhas da matriz resultante são tratadas em função das linhas de a , as colunas da matriz resultante são tratadas em função das colunas de b e a repetição mais interna é controlada de acordo com o número de colunas de a (que é igual ao número de linhas de b).

```

1 import java.util.Scanner;
2 public class MatrizMultiplicacao
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         Scanner ent = new Scanner(System.in);
7         System.out.println("Digite número de linhas e colunas da primeira matriz");
8         int l = ent.nextInt();
9         int m = ent.nextInt();
10        System.out.println("Digite número de colunas da segunda matriz");
11        int n = ent.nextInt();
12        double [][] a = new double[l][m];
13        double [][] b = new double[m][n];
14        double [][] c = new double[l][n];
15        System.out.println("Digite os elemento da matriz a");
16        for(int i=0; i<l; i++) //repetições para preencher a matriz
17            for(int j=0; j<m; j++)
18                a[i][j] = ent.nextDouble();
19
20        System.out.println("Digite os elemento da matriz b");
21        for(int i=0; i<m; i++) //repetições para preencher a matriz
22            for(int j=0; j<n; j++)
23                b[i][j] = ent.nextDouble();
24
25        for(int i=0; i<l; i++) //repetições para multiplicar as matrizes
26            for(int j=0; j<n; j++)
27                for(int k=0; k<m; k++)
28                    c[i][j] += a[i][k] * b[k][j];
29
30        System.out.println("matriz resultante");
31        for(int i=0; i<l; i++)
32        {
33            for(int j=0; j<n; j++)
34                System.out.print(c[i][j] + " ");
35            System.out.println("");
36        }
37    }
38 } //Fim da Função main
39 } //Fim da Classe

```

Figura 7.21: Possível solução para o problema.

Resumo

Nesta aula, você aprendeu a declarar e acessar vetores e matrizes. Viu que os vetores/matrizes são conjuntos de variáveis de um mesmo tipo e que podem ser acessados através do nome do conjunto e de um ou mais índice (de acordo com o número de dimensões), que indicam a posição

de cada variável. Também foi mostrado como compor algoritmos combinando vetores/matrizes, estruturas de repetição e de desvio, a fim de resolver problemas mais complexos.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com arquivos.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T. H. *et al. Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

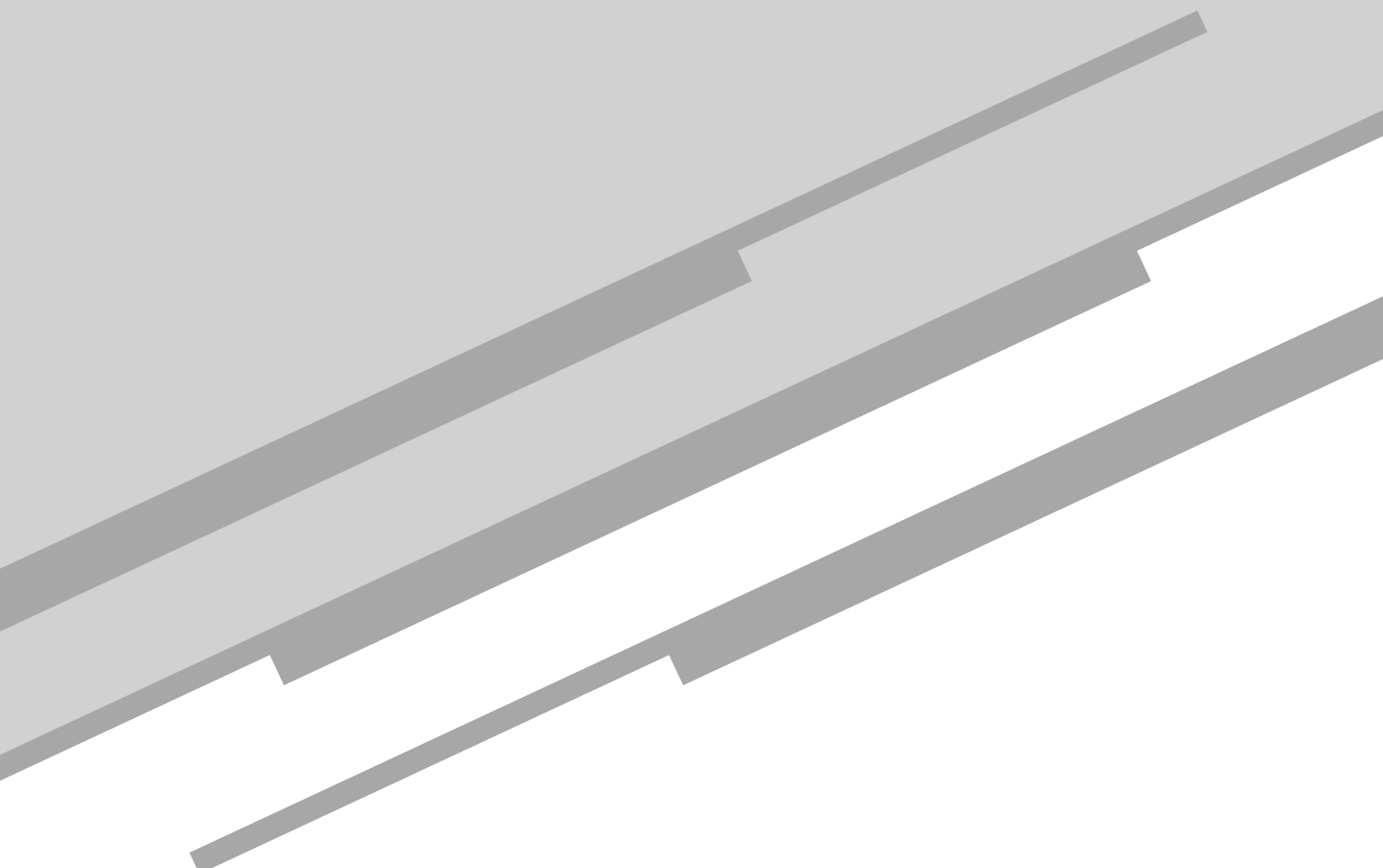
DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. *et al. Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. São Paulo: McGraw-Hill, 1987.

Aula 8

Arquivos



Tiago Araújo Neves

Meta

Expor os conceitos de arquivos, permitindo a construção de programas que utilizem estes recursos.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. entender e aplicar os conceitos de arquivos;
2. construir programas capazes de utilizar estes conceitos.

Introdução

O uso de arquivos em computação é de fundamental importância para a persistência dos dados. Programas que lidam com informações passadas pelo usuário são viáveis e até comerciais em alguns casos, porém, ao reiniciar o computador, estas informações se perdem, o que por vezes é inconveniente. Em alguns casos, a perda destas informações é crítica, fazendo com que o usuário perca o interesse em utilizar o programa, ou ainda inviabilizando o uso, pois o usuário pode não se lembrar de todas as informações digitadas.

Imagine como seria inconveniente tirar uma foto de uma paisagem incrível durante sua viagem de férias e, ao tentar acessá-la, verificar que a foto foi perdida porque a bateria do seu celular se esgotou. O que permite a persistência da foto, ou seja, a existência dela na ausência de energia, é justamente o armazenamento dos dados da imagem em um arquivo.

Arquivos de certas extensões seguem padrões específicos de organização. Como exemplo, podemos citar a organização de um arquivo PDF. Os arquivos PDF são divididos em quatro partes: cabeçalho, corpo, tabela de referência cruzada e *trailer*.

- O cabeçalho indica a versão da especificação PDF sendo usada.
- O corpo contém os objetos a serem exibidos (texto, fotos, etc.) e as especificações de posicionamento e tamanho.
- A tabela de referência cruzada contém referências para todos os objetos no documento. O objetivo de usar esta tabela é que um objeto possa ser acessado sem necessidade de começar uma busca pelo início do documento.
- O *trailer* especifica como uma aplicação de leitura de um arquivo PDF deve encontrar a tabela de referência cruzada e outros objetos especiais do formato PDF.

Note que este tipo de organização implica que os arquivos PDF devem começar a ser lidos pelo seu final, já que o *trailer* deve ser a primeira coisa a ser lida.

O exemplo citado mostra que lidar com arquivos não é uma tarefa trivial em alguns casos. Entretanto, você vai aprender que, apesar das organizações sofisticadas de alguns formatos, é possível criar arquivos de uma maneira relativamente simples e que estes arquivos podem ser utilizados por suas aplicações para facilitar a leitura de dados ou armazenar

resultados importantes. Nesta disciplina, não veremos nenhuma estrutura organizacional de arquivos em particular. Você irá aprender a lidar com arquivos de maneira geral. Caso você tenha interesse em formatos específicos (PDF, DOC, XLS, etc.), você deve procurar por catálogos de referência na Internet.

O que será apresentado nesta disciplina é um modo simples para se lidar com arquivos textuais de forma sequencial, sem se preocupar com a eficiência. Java provê diversos mecanismos para lidar com arquivos textuais e binários de forma sequencial ou direta. Mais informações sobre outros mecanismos de leitura e escrita em arquivos podem ser encontradas na obra de Deitel & Deitel (2012).

Definição e operações

Por definição, um arquivo é uma unidade autocontida de armazenamento de dados, que fica disponível para o sistema operacional e para programas. Mas antes de lidar especificamente com arquivos em Java, é interessante entender um pouco sobre o padrão da linguagem.

Quando variáveis de tipos primitivos (*int*, *double* etc.) se tornam insuficientes para lidar com certas situações, Java permite a criação de classes para que “variáveis” compostas/complexas possam ser criadas e utilizadas. As instâncias destas classes são chamadas de objetos. Neste sentido, quando você precisa armazenar informações primitivas, você deve criar variáveis da natureza da informação desejada. Quando a natureza da informação não é primitiva, em Java devemos criar objetos de classes que representam essa natureza.

Java possui diversas classes predefinidas, criadas para ajudar os programadores em diversas situações corriqueiras. Você já tem lidado com algumas destas classes desde a primeira aula, embora só agora este assunto esteja sendo abordado. As classes *String* e *Scanner* são dois exemplos. Ao executar a linha *Scanner ent = new Scanner(System.in)*, são feitas três coisas:

1. criação de uma referência para um objeto do tipo *Scanner* chamada *ent*;
2. criação de um objeto do tipo *Scanner* através do comando *new*;
3. associação do objeto à referência através do operador de atribuição.

Outro exemplo de objetos que você já utilizou são os *arrays*. Para situações não corriqueiras, Java permite que o programador crie novas classes, mas por enquanto vamos nos ater às classes predefinidas. Este assunto será abordado novamente em momento oportuno.

Agora que você entende um pouco mais do padrão Java em relação à criação de objetos, podemos começar o estudo de como lidar com arquivos nesta linguagem. Como pode ser notado, um arquivo é uma entidade com especificidades que não podem ser expressas por nenhum tipo primitivo, e como lidar com arquivos é uma operação corriqueira dentro da computação, Java possui uma classe, chamada *File*, para lidar com esta entidade. A linha a seguir mostra um exemplo de criação de um objeto desta classe:

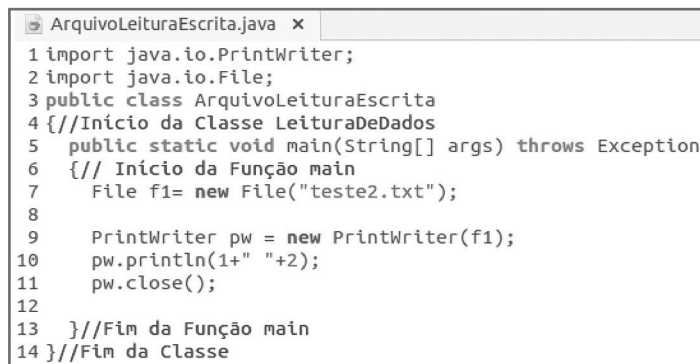
```
File f1= new File("teste2.txt");
```

Esta linha cria um objeto da classe *File* ligado ao arquivo de nome *teste2.txt* e atribui o objeto criado à referência *f1*. Uma vez feita esta associação, é possível verificar se o arquivo existe, se ele é um arquivo ou diretório, entre outras operações, todas através da referência *f1*.

Algumas das operações de arquivos são triviais e podem ser feitas através de qualquer explorador. Você possivelmente já lidou com algumas delas, como *criar*, *remover* e *renomear*. Dentro do contexto da programação, quatro operações merecem atenção especial. São elas:

- abertura,
- leitura,
- escrita e
- fechamento.

Para ajudar na compreensão destas operações vejamos um exemplo completo para a escrita em arquivo, exposto pela **Figura 8.1**. Ao executar este programa, você verá que há no diretório um arquivo chamado *teste2.txt* e, ao abri-lo, você verá que o texto "1 2" está armazenado nele.



```

ArquivoLeituraEscrita.java x
1 import java.io.PrintWriter;
2 import java.io.File;
3 public class ArquivoLeituraEscrita
4 { //Início da Classe LeituraDeDados
5     public static void main(String[] args) throws Exception
6     { // Início da Função main
7         File f1= new File("teste2.txt");
8
9         PrintWriter pw = new PrintWriter(f1);
10        pw.println(1+" "+2);
11        pw.close();
12
13    } //Fim da Função main
14 } //Fim da Classe

```

Figura 8.1: Exemplo de escrita em arquivo.

Operação de abertura – Um arquivo é um recurso externo ao programa, logo, para ter acesso a este recurso, o programa deve fazer uma solicitação ao sistema operacional, que gerencia todos os recursos do computador. Um dos diversos modos de se fazer isso em Java é associar o arquivo a um objeto de leitura ou escrita. No exemplo da **Figura 8.1**, esta operação é feita pela linha 9:

```
PrintWriter pw = new PrintWriter(f1);
```

Nesta linha, é criado um objeto da classe *PrintWriter*, associado ao arquivo *f1*, criado anteriormente, e este objeto é associado à referência *pw*. No momento da criação do objeto *PrintWriter*, o arquivo associado a *f1* é aberto para a escrita. Para ter acesso às classes *PrintWriter* e *File*, é preciso importá-las. Isto é feito através das linhas 1 e 2 do programa.

Operações de leitura e escrita – São operações utilizadas para extrair e inserir informações dos arquivos. Têm a mesma finalidade que as operações de leitura e escrita vistas até o momento. No exemplo da **Figura 8.1** é feita uma operação de escrita na linha 10:

```
pw.println(1+" "+2);
```

Nesta linha, o resultado da expressão *1+" "+2* é impresso na saída ligada ao objeto referenciado por *pw*, que é o arquivo *f1*, que por sua vez está associado ao arquivo *teste2.txt*.

Operação de fechamento – Informa ao sistema operacional que a associação entre o arquivo e o programa está desfeita, fazendo com que o sistema operacional tome as medidas necessárias (por exemplo, verificar se os dados foram escritos, liberar o acesso ao arquivo para outros

programas etc.). O modo de fazer esta operação usando um objeto do tipo *PrintWriter* é através de seu método *close*. A linha a seguir mostra uma chamada do método *close* para o objeto referenciado por *pw* definido anteriormente.

```
pw.close();
```

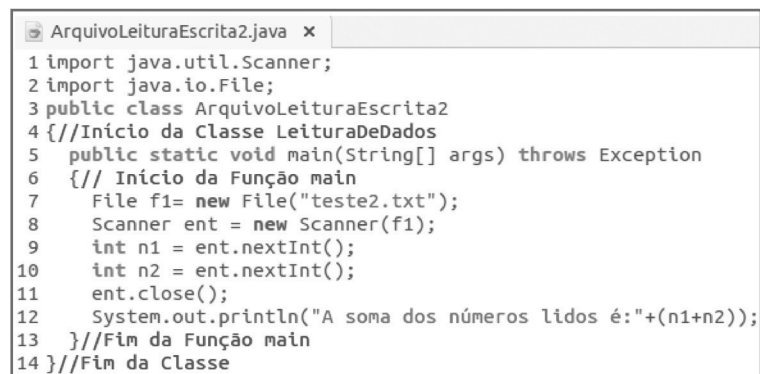
Observe que foi necessária uma modificação na assinatura da função *main*. Agora, após a lista de parâmetros, existe um “*throws Exception*”. Java possui um sofisticado sistema de tratamento de erros e exceções. Em alguns casos, para usarmos determinadas funcionalidades de algumas classes, é obrigatório o uso deste mecanismo ou uma informação explícita de que a exceção ocorrida será repassada. O uso de arquivos é um destes casos e, como você ainda não estudou este mecanismo de tratamento de exceções, vamos apenas informar que a função *main* irá repassar a exceção ocorrida acrescentando *throws Exception* após a lista de parâmetros.



A diretiva *throws*

A diretiva *throws*, em Java, indica que a função ou método pode lançar exceções. O ato de uma função *a1* poder lançar uma exceção não necessariamente implica que a exceção ocorre em *a1*. Em alguns casos, a exceção ocorre em chamadas de outras funções dentro de *a1*. Nestes casos, a diretiva *throws* colocada em *a1* indica que as exceções ocorridas simplesmente serão repassadas. Este repasse pode indicar que a exceção não será tratada, ou será tratada por outra função, por exemplo, a função que invocou *a1*. A rigor, todas as exceções podem ser tratadas. Quando não são tratadas de modo explícito pelo código, elas são repassadas para a máquina virtual, que geralmente as trata através da impressão das informações sobre a exceção ocorrida e do encerramento da aplicação.

É importante citar que dados em arquivos não podem ser processados diretamente. Isto quer dizer que, se você quer somar dois números que estão armazenados em arquivos, por exemplo, primeiro é necessário ler estes dois números, armazená-los em duas variáveis, e então executar a soma. Para fazer leituras em arquivos, é necessário apenas ligar um objeto *Scanner* ao arquivo de modo correto. A **Figura 8.2** mostra um exemplo completo de um programa que faz leitura de dados em arquivos. Note que o objeto *Scanner* agora não está mais associado à entrada padrão do sistema (*System.in*), e sim ao objeto referenciado por *f1*. Como *f1* está ligado ao arquivo *teste2.txt*, criado pelo programa da **Figura 8.1**, o objeto *Scanner* irá ler os dados do disco.



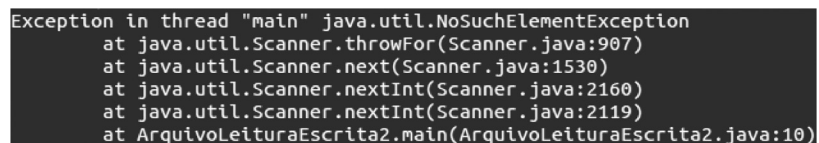
```

1 import java.util.Scanner;
2 import java.io.File;
3 public class ArquivoLeituraEscrita2
4 { //Início da Classe LeituraDeDados
5     public static void main(String[] args) throws Exception
6     { // Início da Função main
7         File f1= new File("teste2.txt");
8         Scanner ent = new Scanner(f1);
9         int n1 = ent.nextInt();
10        int n2 = ent.nextInt();
11        ent.close();
12        System.out.println("A soma dos números lidos é:"+(n1+n2));
13    } //Fim da Função main
14 } //Fim da Classe

```

Figura 8.2: Exemplo de leitura de dados armazenados em arquivo.

Alguns erros são cometidos com frequência ao lidar com arquivos. Um deles ocorre quando tentamos ler mais dados do que o arquivo possui. Neste caso, ao retirarmos o último número do arquivo *teste2.txt* e executarmos o programa da **Figura 8.2**, obtemos o erro exposto pela **Figura 8.3**. Neste caso, o erro ocorre na linha 10 do programa *ArquivoLeituraEscrita2.java*, que é exatamente quando tentamos ler o segundo número de um arquivo que possui apenas um.



```

Exception in thread "main" java.util.NoSuchElementException
    at java.util.Scanner.throwFor(Scanner.java:907)
    at java.util.Scanner.next(Scanner.java:1530)
    at java.util.Scanner.nextInt(Scanner.java:2160)
    at java.util.Scanner.nextInt(Scanner.java:2119)
    at ArquivoLeituraEscrita2.main(ArquivoLeituraEscrita2.java:10)

```

Figura 8.3: Erro ao ler dados a mais do que o arquivo possui.

Outro erro frequente é tentar ler dados de um arquivo que não existe. Este erro é exposto pela **Figura 8.4**. Para este teste, o arquivo *teste2.txt* foi removido do diretório, e em seguida o programa *ArquivoLeitura-Escrita2* foi executado. Note que o erro ocorre somente ao tentar abrir o arquivo para leitura (linha 8), e não ao criar a referência para o arquivo (linha 7).

```
Exception in thread "main" java.io.FileNotFoundException: teste2.txt (No such file or directory)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:146)
    at java.util.Scanner.<init>(Scanner.java:656)
    at ArquivoLeituraEscrita2.main(ArquivoLeituraEscrita2.java:8)
```

Figura 8.4: Erro ao ler dados de um arquivo inexistente.

Conclusão

Uma vez que você já sabe como armazenar dados de maneira persistente, cabe a você definir quando utilizar este recurso de maneira conveniente. Ficar digitando dados de matrizes a cada execução é uma tarefa tediosa e sujeita a erros. Imagine que você quer testar um algoritmo de multiplicação de duas matrizes 10x10 e que você está inserindo os números manualmente para a execução do programa. Se um dos valores for digitado errado, todo o processo deve ser reiniciado. Neste caso, é evidente que o uso de arquivos pode trazer contribuições. Utilizar arquivos para salvar resultados importantes também pode ser de grande utilidade para que você não tenha que ficar memorizando respostas.

Atividade

Atende aos objetivos 1 e 2

1.
 - a) Faça um programa para salvar em arquivo os valores de um vetor de números inteiros digitados pelo usuário. O tamanho deve ser informado pelo usuário e armazenado no começo do arquivo. Cada número deve ser armazenado em uma linha do arquivo. Ex.: suponha o vetor de tamanho 3 com os valores 5, 6, 7. O arquivo deve conter os números 3, 5, 6 e 7 dispostos em linhas diferentes, neste caso o arquivo ficaria assim:

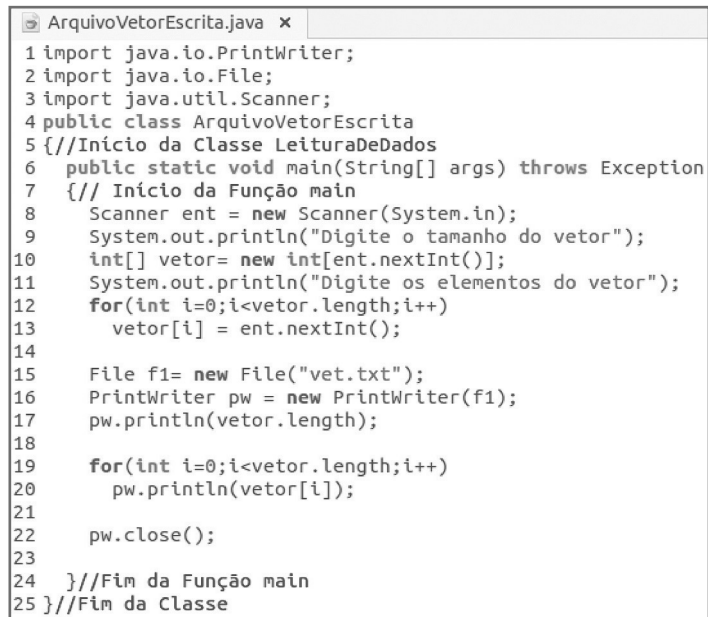
[illegible]

b) Faça um programa capaz de ler de um arquivo os dados de uma matriz dispostos como descrito no item anterior. O nome do arquivo a ser lido deve ser informado pelo usuário. Os dados lidos devem ser impressos em tela.

[illegible]**Resposta comentada**

1.

a) Para resolver este problema, o primeiro passo é ler os dados do vetor. Em seguida deve-se criar um arquivo, criar um objeto *PrintWriter* associado a este arquivo e escrever o tamanho do vetor usando o método *println*. Na sequência, os valores contidos no vetor devem ser escritos no arquivo, também com o uso do método *println*. A seguir, encontra-se uma possível solução.



```

1 import java.io.PrintWriter;
2 import java.io.File;
3 import java.util.Scanner;
4 public class ArquivoVetorEscrita
5 { //Início da Classe LeituraDeDados
6     public static void main(String[] args) throws Exception
7     { // Início da Função main
8         Scanner ent = new Scanner(System.in);
9         System.out.println("Digite o tamanho do vetor");
10        int[] vetor= new int[ent.nextInt()];
11        System.out.println("Digite os elementos do vetor");
12        for(int i=0;i<vetor.length;i++)
13            vetor[i] = ent.nextInt();
14
15        File f1= new File("vet.txt");
16        PrintWriter pw = new PrintWriter(f1);
17        pw.println(vetor.length);
18
19        for(int i=0;i<vetor.length;i++)
20            pw.println(vetor[i]);
21
22        pw.close();
23
24    } //Fim da Função main
25 } //Fim da Classe

```

Figura 8.5: Para a solução, o primeiro passo é ler os dados do vetor e, em seguida, criar um arquivo e um objeto *PrintWriter* associado a este e escrever o tamanho do vetor usando o método *println*.

b) Este exercício requer um pouco mais de atenção. Note que é pedido para que o arquivo do item anterior seja utilizado. Isto implica que o tamanho do vetor deve ser lido também, o que por sua vez requer que a alocação do vetor seja feita após a primeira leitura. A primeira coisa a ser feita é ler o nome do arquivo. Para isso, deve-se utilizar um objeto *Scanner* ligado ao teclado (*System.in*).

Os próximos passos são criar um objeto *File* utilizando o nome lido e criar um segundo objeto *Scanner*, ligado ao *File* criado. Se tudo ocorrer da maneira correta, isso irá abrir o arquivo desejado para leitura dos dados. O próximo passo é ler somente a primeira informação vinda do arquivo, que é o tamanho do vetor. Com este tamanho, pode-se alocar o vetor e controlar de forma correta as repetições de leitura dos dados no arquivo. Por último, deve-se ler os demais dados do arquivo. A seguir está uma possível solução.

```

ArquivoVetorLeitura.java x
1 import java.io.File;
2 import java.util.Scanner;
3 public class ArquivoVetorLeitura
4 { //Início da Classe LeituraDeDados
5     public static void main(String[] args) throws Exception
6     { // Início da Função main
7         Scanner ent = new Scanner(System.in);
8         System.out.println("Digite o nome do arquivo que contem o vetor");
9         String nome = ent.next();
10        File f1= new File(nome);
11        Scanner arq = new Scanner(f1);
12        int[] vetor= new int[arq.nextInt()];
13
14        for(int i=0;i<vetor.length;i++)
15            vetor[i] = arq.nextInt();
16
17        arq.close();
18
19        System.out.println("Os dados do arquivo são:");
20        for(int i=0;i<vetor.length;i++)
21            System.out.println(vetor[i]);
22
23    } //Fim da Função main
24 } //Fim da Classe

```

Figura 8.6: A primeira coisa a ser feita é ler o nome do arquivo. Para isso, deve-se utilizar um objeto *Scanner* ligado ao teclado (*System.in*).

2.

a) Assim como no item *a* da atividade anterior, a primeira coisa a ser feita é ler os dados da matriz. Uma vez feito isso, deve-se criar um arquivo a partir do nome informado pelo usuário e criar um objeto *PrintWriter* associado ao arquivo. Em seguida, para escrever o número de linhas e colunas na mesma linha, pode-se utilizar um único comando *println*. Na sequência, devem-se utilizar comandos de repetição de modo conveniente para que os números da mesma linha da matriz sejam armazenados na mesma linha do arquivo. A seguir está uma possível solução.

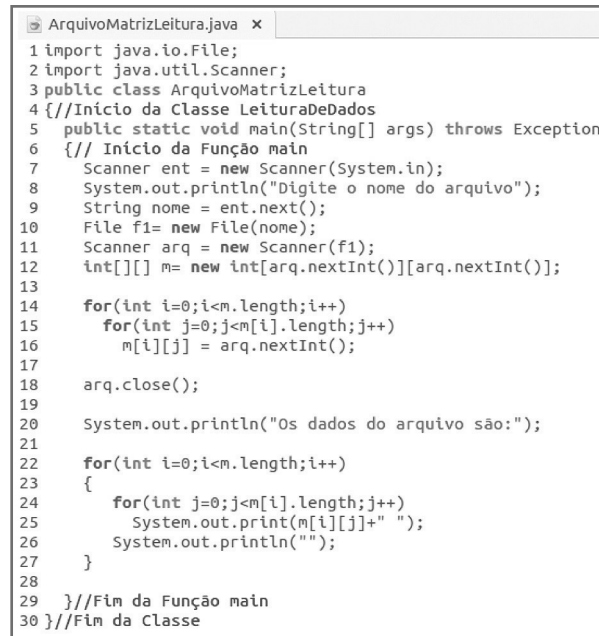
```

ArquivoMatrizEscrita.java x
1 import java.io.PrintWriter;
2 import java.io.File;
3 import java.util.Scanner;
4 public class ArquivoMatrizEscrita
5 { //Início da Classe LeituraDeDados
6     public static void main(String[] args) throws Exception
7     { // Início da Função main
8         Scanner ent = new Scanner(System.in);
9         System.out.println("Digite o número de linhas e colunas da matriz");
10        int[][] matriz= new int[ent.nextInt()][ent.nextInt()];
11
12        System.out.println("Digite os elementos da matriz");
13        for(int i=0;i<matriz.length;i++)
14            for(int j=0;j<matriz[i].length;j++)
15                matriz[i][j] = ent.nextInt();
16
17        System.out.println("Digite o nome do arquivo");
18        String nome = ent.next();
19        File f1= new File(nome);
20        PrintWriter pw = new PrintWriter(f1);
21        pw.println(matriz.length+" "+matriz[0].length);
22
23        for(int i=0;i<matriz.length;i++)
24        {
25            for(int j=0;j<matriz[i].length;j++)
26                pw.print(matriz[i][j]+" ");
27            pw.println("");
28        }
29        pw.close();
30
31    } //Fim da Função main
32 } //Fim da Classe

```

Figura 8.7: A primeira coisa a ser feita é ler os dados da matriz e, então, criar um arquivo a partir do nome informado pelo usuário e um objeto *PrintWriter* associado ao arquivo.

b) Assim como no item *b* da atividade anterior, este exercício requer o uso de dois objetos *Scanner*. Também, devem-se ler informações referentes à alocação do arquivo para só então ler os valores da matriz em si. Por último, deve-se imprimir na tela o conteúdo do arquivo. A seguir está uma possível solução.



```

1 import java.io.File;
2 import java.util.Scanner;
3 public class ArquivoMatrizLeitura
4 { //Início da Classe LeituraDeDados
5     public static void main(String[] args) throws Exception
6     { // Início da Função main
7         Scanner ent = new Scanner(System.in);
8         System.out.println("Digite o nome do arquivo");
9         String nome = ent.next();
10        File f1= new File(nome);
11        Scanner arq = new Scanner(f1);
12        int[][] m= new int[arq.nextInt()][arq.nextInt()];
13
14        for(int i=0;i<m.length;i++)
15            for(int j=0;j<m[i].length;j++)
16                m[i][j] = arq.nextInt();
17
18        arq.close();
19
20        System.out.println("Os dados do arquivo são:");
21
22        for(int i=0;i<m.length;i++)
23        {
24            for(int j=0;j<m[i].length;j++)
25                System.out.print(m[i][j]+" ");
26            System.out.println("");
27        }
28
29    } //Fim da Função main
30 } //Fim da Classe

```

Figura 8.8: Este exercício requer o uso de dois objetos *Scanner*, para só então ler informações referentes à alocação do arquivo e, conseqüentemente, ler os valores da matriz em si.

Resumo

Nesta aula você aprendeu a lidar com armazenamento de dados em arquivos. Aprendeu, na primeira seção, que tipos diferentes de arquivos possuem diferentes tipos de organização, sendo alguns tipos muito sofisticados (como os arquivos de formato PDF) e outros muito simples (como os arquivos TXT). Também foi mostrado como construir programas que utilizam arquivos para leitura e escrita sequenciais.

Informação sobre a próxima aula

Na próxima aula, você aprenderá a lidar com funções.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T. H. *et al. Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

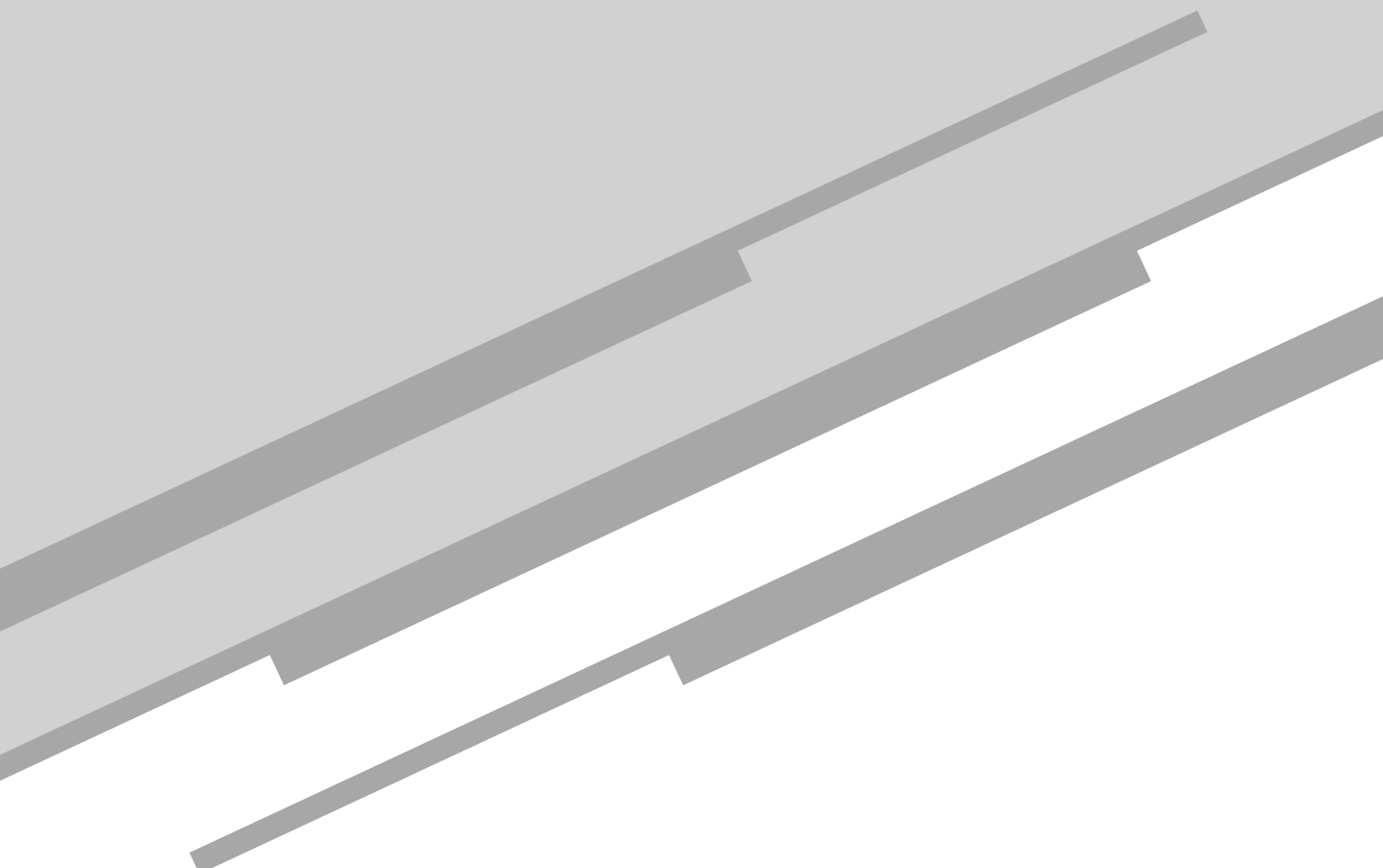
DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. *et al. Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. São Paulo: McGraw-Hill, 1987.

Aula 9

Funções – Parte I



Meta

Expor os conceitos básicos de funções, mostrando sua sintaxe e funcionamento.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. definir e aplicar o conceito de funções;
2. fazer programas que utilizam funções para resolver problemas.

Introdução

Em muitos programas feitos até o momento, mesmo fazendo uso de estrutura de repetição, é necessário fazer replicação do código. Um exemplo claro são os algoritmos que envolvem matrizes. Quando é preciso fazer a leitura de dados de uma matriz de duas dimensões, são feitas duas estruturas de repetição: uma para as linhas e outra para as colunas. Dentro destas duas estruturas, fica a instrução de leitura da matriz nas posições controladas pelas repetições. Para ler uma segunda matriz, deve-se criar um segundo conjunto com duas estruturas de repetição. Para ler uma terceira matriz, um terceiro conjunto, e assim sucessivamente.

Este padrão de repetições sucessivas de conjuntos muito semelhantes de estruturas acaba por aumentar em demasia o tamanho dos programas. Além disso, fazer manutenção (corrigir erros) em programas que seguem este modelo é uma tarefa árdua, uma vez que, se um dos conjuntos de repetição deve ser alterado, provavelmente muitos outros também deverão ser.

Para lidar com este tipo de problema (repetições de comportamento para entidades diferentes) dentro da computação, são utilizadas **funções** e **procedimentos**. Tanto as funções quanto os procedimentos são **subprogramas**, e podem ser definidos como trechos de códigos parametrizados, invocáveis a partir de um nome, que executam uma ou mais tarefas computacionais. A ideia é similar às funções da matemática.

Tome como exemplo a função $f(x) = 2x + 1$. Esta função pode ser aplicada a qualquer valor de x pertencente a \mathbb{R} , e o comportamento será sempre o mesmo, o valor de x será dobrado e depois acrescido de uma unidade. Deste modo, o que as funções fazem é estabelecer comportamentos que serão executados a elementos de um conjunto de dados de entrada. Note que, para diferentes valores de x , têm-se diferentes resultados. Por exemplo: para $x = -2$, tem-se $f(x) = -3$; para $x = 2$, tem-se $f(x) = 5$. Observe que, dependendo do valor do parâmetro de entrada, a função apresenta um resultado diferente, mas o comportamento é sempre o mesmo. O parâmetro de entrada é dobrado e acrescido de uma unidade.

Em Java, o conceito de funções é usado com frequência, entretanto o conceito de procedimento não existe de forma explícita. Os procedimentos são implementados através de funções especiais, que serão vistas em momento oportuno. Nesta aula, você aprenderá como construir funções em Java, e que o uso de funções pode simplificar muito os programas, bem como facilitar sua manutenção. Além disso, o uso de

Subprograma

Conjunto parametrizado de instruções invocáveis através de um nome que executam uma ou mais tarefas computacionais.

Função

Subprograma com retorno obrigatório. Os resultados de funções podem ser usados para compor expressões de maneira direta.

Procedimento

Subprograma sem retorno obrigatório, também chamado de sub-rotina.

funções promove, de maneira geral, a redução do código-fonte, o que facilita a construção e o entendimento de algoritmos/programas para problemas complexos.

Funções: declaração e uso

A declaração de funções em Java segue a sintaxe da **Figura 9.1**. Os modificadores são parte da sintaxe Java e estão principalmente relacionados ao acesso de membros de classe, ligados à programação orientada a objetos. Como os conceitos deste tipo de programação não são o foco desta aula, usaremos sempre os mesmos dois modificadores: *public static*.

Conceitos de orientação a objetos serão discutidos em momento oportuno ou em outras disciplinas do curso. Neste sentido, todas as funções feitas por enquanto deverão conter estes dois modificadores. Após estes modificadores, encontra-se o tipo da função. Este tipo é a natureza do dado que será a resposta da função. Exemplo: se uma função é do tipo *int*, a resposta que ela retornará será um número inteiro; se ela for do tipo *String*, ela retornará um texto, etc.

```
modificadores tipo nome (<lista de parâmetros>)
{
    <Instruções>
}
```

Figura 9.1: Sintaxe de declaração de funções.

Após o tipo, encontra-se o “nome” da função, que nada mais é do que um identificador válido que será associado ao trecho de código. Os nomes dados a funções podem ser quaisquer identificadores válidos; contudo, normalmente utilizam-se identificadores que expressam o que a função irá fazer. A matemática tem por hábito nomear as funções com letras, por exemplo, função *f*, função *g*, etc. Embora isso também seja possível, na computação esta prática não é bem vista por cientistas e programadores. Um bom nome para uma função deve dar uma ideia clara do que ela faz, sem que seja necessário olhar toda a estrutura da função.

Suponha que existam duas funções: uma função chamada *a1* e outra chamada *raizQuadrada*. Note que só pelo nome já é possível deduzir o que a função *raizQuadrada* irá fazer, o que não ocorre com a função *a1*. Neste caso, para saber o que a função *a1* faz, é necessário olhar para toda a sua estrutura.

Os parâmetros são passados com tipo e nome separados por vírgulas.

Um possível exemplo de lista de parâmetros é o seguinte: *int idade*, *double salario*, *String nome*. Em alguns casos, podem existir parâmetros do mesmo tipo: *int idade*, *int numFilhos*, *int numFilhas*. Nestes casos, deve-se especificar o tipo de cada parâmetro, mesmo que eles tenham o mesmo tipo. É importante ressaltar que o tipo dos parâmetros nada tem a ver com o tipo de resposta de uma função. Por exemplo, pode haver funções que recebem números reais como parâmetros e retornam números reais. Entretanto, também pode haver funções que recebem textos como parâmetros e retornam números como resposta. Um exemplo seria uma função para calcular quantas vezes uma determinada letra aparece em uma palavra.

Os delimitadores de bloco (`{` e `}`) são obrigatórios e entre eles estão as instruções que compõem o corpo da função. Quaisquer instruções podem ser utilizadas: criação de variáveis e vetores, desvios condicionais, estruturas de repetição, etc. A única exceção é a declaração de funções.

Ou seja, dentro de uma função, podem existir quaisquer instruções de código, exceto a definição de outras funções. Para criar mais de uma função, deve-se fazê-las separadamente. Assim, para fazer três funções, por exemplo, é necessário que elas estejam dispostas separadamente, uma após a outra.

O uso de funções em programas

Para ilustrar o uso de funções, vejamos a solução de um problema muito comum no dia a dia de um engenheiro: o problema de conversão de unidades. Sabe-se que um centímetro corresponde a 0,3937 polegadas. Devem-se imprimir as polegadas que correspondem de 0, 0.5, 1.0, ..., 10 centímetros. A **Figura 9.2** mostra a função utilizada para este problema. Ela converte um valor *double* passado como parâmetro (*cm*) para seu correspondente em polegadas através de uma regra de três simples. Note que dentro da função existe a instrução *return*. Esta instrução indica que o que está escrito à frente dela deve ser retornado como resultado da função. O uso desta instrução é obrigatório e ela é sempre a última instrução da linha a ser executada. Assim, primeiramente, será feita a multiplicação da variável *cm* com a constante 0.3937. Em seguida, o valor da multiplicação é retornado como resultado da função.

```
public static double converterCmParaPol(double cm)
{
    return 0.3937*cm;
}
```

Figura 9.2: Função para converter centímetros em polegadas.

A **Figura 9.3** mostra um programa que faz uso da função *converterCmParaPol* descrita na **Figura 9.2** para resolver o problema citado. Note que a função é chamada diversas vezes dentro da estrutura de repetição e que seu resultado é impresso diretamente.

```
FuncaoExemplo.java x
1 public class FuncaoExemplo
2 { //Início da Classe
3     //função para converter
4     public static double converterCmParaPol(double cm)
5     { //Início da Função converterCmParaPol(double cm)
6         return 0.3937*cm;
7     } //Fim da Função converterCmParaPol(double cm)
8
9     public static void main(String[] args)
10    { //Início da Função main
11        for(int i=0; i<=20; i++)
12        {
13            System.out.println(converterCmParaPol(i*0.5));
14        }
15    } //Fim da Função main
16 } //Fim da Classe
17
```

Figura 9.3: Programa que usa a função definida na **Figura 9.2**.

Antes de começarmos as atividades é necessário esclarecer alguns pontos. O primeiro é a posição das funções em relação à função *main*. Este posicionamento varia muito de acordo com a linguagem de programação. Em Java, não existem restrições de posicionamento em relação à função *main*. As funções podem ser posicionadas antes (caso da **Figura 9.3**), ou depois. O único cuidado a ser tomado é que todas as funções devem ser definidas dentro da classe.



Mesma lógica, diferentes formas

Em C/C++ é usual colocar as funções antes do programa que faz as chamadas (embora existam mecanismos para burlar isso).

Em Fortran, as funções/subrotinas normalmente são colocadas depois do programa que faz a chamada.

Outro ponto a esclarecer é como funciona a chamada de uma função em Java. As chamadas funcionam exatamente como visto na disciplina de Computação I. O fluxo do programa começa pela função *main* e segue normalmente até a chamada da função. Neste momento ocorre um desvio de fluxo para dentro da função invocada, fazendo com que as instruções do corpo desta função sejam executadas. Quando a função termina sua execução, o fluxo é novamente desviado para o ponto onde foi feita a chamada. É importante mencionar que as variáveis declaradas dentro de funções são criadas no momento da chamada e destruídas quando a função termina.



A função só pode ser chamada com a quantidade, a ordem e o tipo exatos de parâmetros estipulados em sua definição.

É importante mencionar que, no momento de definir uma função, o número de parâmetros, o tipo de cada um deles e a ordem em que são dispostos são de extrema importância. Ao definir uma função com três parâmetros, ela não pode ser chamada com apenas dois ou com quatro parâmetros.

No nosso exemplo, o procedimento *converterCmParaPol* só pode ser chamado com um parâmetro. O tipo de cada um dos parâmetros também é de fundamental importância. Se, na definição de uma função, for passado como parâmetro um vetor de números reais, não adianta passar um vetor de números inteiros no momento da chamada. Ainda no nosso exemplo, a função criada só recebe números reais. A ordem dos parâmetros estabelecida no momento da definição da função também deve ser respeitada. Logo, se no momento da definição da função, criarmos a seguinte lista de parâmetros (*int a, double b*) não é possível chamar a função passando o número *double* primeiro e depois o *int*. Isso ocorre porque, no momento da definição, estabeleceu-se que o primeiro parâmetro é sempre o *int* e o segundo é sempre o *double*.

===== **Atividade 1** =====

Atende ao objetivo 1

a) Faça uma função que receba três números reais. A função deve determinar se estes números formam ou não um triângulo. Obs: Três números reais podem representar os lados de um triângulo se, e somente se, a soma de quaisquer dois deles for maior que o terceiro.

b) Faça uma função que receba como parâmetro um vetor (matriz unidimensional) de números inteiros. Esta função deve calcular a média dos elementos deste vetor.

Resposta comentada

a) O enunciado já determinou a quantidade e a natureza dos parâmetros da função. O tipo de retorno, como equivale a responder sim ou não, pode ser *boolean*. Para determinar se três números formam ou não um triângulo, basta analisar se a soma de quaisquer dois deles é maior que o terceiro. Uma possível solução é exibida a seguir.

```
public static boolean eTriangulo(double a, double b, double c)
{
    return a+b>c && a+c >b && b+c>a;
}
```

Figura 9.4: Possível solução para o problema.

b) O enunciado da questão diz que o vetor deve ser passado como parâmetro, portanto, ao analisar a situação, você perceberá que nenhum outro parâmetro é necessário. Para calcular a média de um vetor, é necessário saber qual vetor será usado, seu tamanho, uma variável de contagem, uma variável para armazenar a soma e uma estrutura de repetição. Tendo o vetor como parâmetro e seu tamanho acessível pelo membro *length*, a variável de contagem e a de soma podem ser declaradas como variáveis internas da função. Uma possível solução é mostrada a seguir.

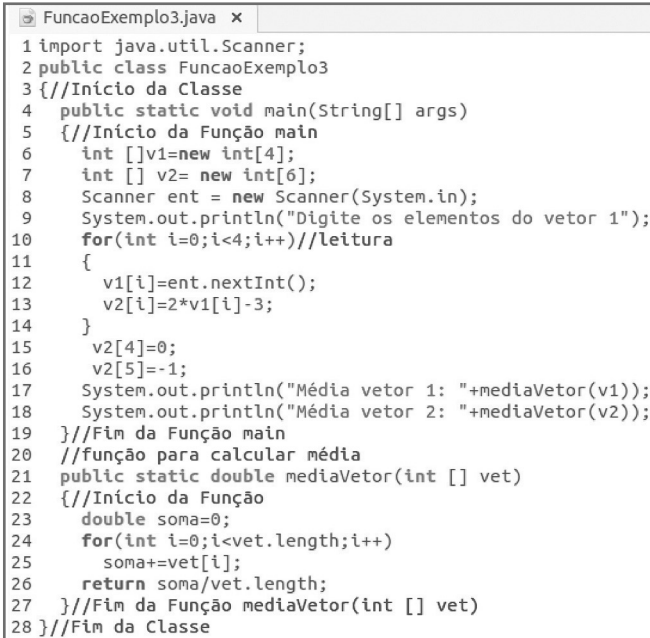
```
public static double mediaVetor(int [] vet)
{ //Início da Função
    double soma=0;
    for(int i=0;i<vet.length;i++)
        soma+=vet[i];
    return soma/vet.length;
} //Fim da Função mediaVetor(int [] vet)
```

Figura 9.5: Possível solução para o problema.

Lembrando que, mesmo que os números sejam inteiros, a média pode ser um número real.

Um ponto importante a ser mencionado é que uma mesma função pode ser usada para qualquer conjunto de parâmetros que atenda os requisitos de sua lista. A **Figura 9.6** mostra um exemplo disso. Nela são declarados dois vetores de tamanhos diferentes, preenchidos com dados diferentes, e a mesma função sendo usada para calcular as médias dos dois vetores. Este é exatamente um dos objetivos de se utilizar funções: repetir padrões de comportamento para dados diferentes.

O uso de nomes diferentes é possível porque, no momento da chamada da função, é criada uma referência (um segundo nome) para os dados. No nosso exemplo, na primeira chamada da função, é criada uma referência para *v1* chamada *vet*. Em outras palavras, *vet* pode ser visto como um “segundo nome” para *v1* e, dentro da função, o vetor é tratado por este segundo nome. Veremos este mecanismo em detalhes em um momento oportuno. Por enquanto, basta saber que uma função pode ser chamada usando qualquer conjunto de variáveis/vetores que se encaixe na lista de parâmetros de sua definição.



```

FuncaoExemplo3.java x
1 import java.util.Scanner;
2 public class FuncaoExemplo3
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         int [] v1=new int[4];
7         int [] v2= new int[6];
8         Scanner ent = new Scanner(System.in);
9         System.out.println("Digite os elementos do vetor 1");
10        for(int i=0;i<4;i++)//leitura
11        {
12            v1[i]=ent.nextInt();
13            v2[i]=2*v1[i]-3;
14        }
15        v2[4]=0;
16        v2[5]=-1;
17        System.out.println("Média vetor 1: "+mediaVetor(v1));
18        System.out.println("Média vetor 2: "+mediaVetor(v2));
19    } //Fim da Função main
20    //função para calcular média
21    public static double mediaVetor(int [] vet)
22    { //Início da Função
23        double soma=0;
24        for(int i=0;i<vet.length;i++)
25            soma+=vet[i];
26        return soma/vet.length;
27    } //Fim da Função mediaVetor(int [] vet)
28 } //Fim da Classe
  
```

Figura 9.6: Reutilização da função para calcular a média de vetores diferentes.

Funções do tipo *Void*

Imagine a seguinte situação: você tem que fazer a impressão de vários vetores do mesmo tipo e, para isso, tem que fazer uma função. A **Figura 9.7** mostra uma tentativa de resolução deste problema. Nela, uma função do tipo *int* é construída. Esta função imprime os elementos e, no final, retorna zero.

```
public static int imprimirVetor(int [] vet)
{
    //Início da Função
    for(int i=0;i<vet.length;i++)
        System.out.println(vet[i]);
    return 0;
}
//Fim da Função imprimirVetor(int [] vet)
```

Figura 9.7: Função do tipo *int* para impressão.

Ao analisar esta solução, claramente percebe-se que ela possui um retorno inadequado. Não faz sentido construir uma função que sempre retorne o mesmo valor, independentemente da situação, bem como não faz sentido retornar valores quando nenhum retorno é necessário. Na disciplina Computação I, você aprendeu que, para estes casos, as funções não são a melhor opção e que procedimentos devem ser usados nestas situações. Java não permite explicitamente a criação de procedimentos, mas permite criar funções que não têm retorno, o que equivale a um procedimento. Estas funções possuem um “tipo” de retorno especial, chamado tipo *void*. Uma função do tipo *void* não é obrigada a utilizar a instrução *return*, o que permite, portanto, que ela não retorne nada. A **Figura 9.8** mostra uma solução para o problema mencionado usando uma função do tipo *void*. Note que não há instrução de retorno no corpo desta função.

```
public static void imprimirVetor(int [] vet)
{
    //Início da Função
    for(int i=0;i<vet.length;i++)
        System.out.println(vet[i]);
}
//Fim da Função imprimirVetor(int [] vet)
```

Figura 9.8: Função do tipo *void* para impressão.

Não há diferenças de uso para funções do tipo *void*. A única diferença entre estas funções e as funções convencionais é a ausência de retorno. No mais, tanto na compilação como na execução, funções do tipo *void* são tratadas de maneira exatamente igual às funções convencionais.

A função *main*

Observe a seguinte linha de código:

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

Como você já deve ter percebido, esta é uma declaração de função. A função *main*, presente em todos os programas Java, é uma função especial, sem retorno, e que recebe como parâmetro um vetor de textos. Todo programa Java sempre começa pela execução da função *main*. Isto implica que, não importa que ela seja declarada por último, o fluxo de execução sempre começa por ela e dela é desviado para outras funções, quando necessário. Portanto, toda vez que for necessário construir um programa Java, é obrigatória a construção de uma função *main*.

Funções com múltiplas instruções de retorno

O fluxo de execução dentro de funções normais não tem o mesmo comportamento do que o fluxo dentro funções *void*. Em funções *void*, o fluxo vai do início ao fim. Em funções normais, ele vai do início até encontrar a primeira instrução *return*. A **Figura 9.9** mostra um exemplo de função com duas instruções *return*. Esta função calcula o valor absoluto de um número dado pela equação a seguir:

$$|x| = \begin{cases} x, & \text{se } x \geq 0 \\ -x, & \text{c. c.} \end{cases}$$

A equação diz que o valor absoluto de um número *x*, também chamado de módulo de *x*, é o próprio *x*, se *x* for maior ou igual a zero ou *x* multiplicado por *-1*, caso contrário.

```

public static double valorAbsoluto(double x)
{ //Início da Função
  if(x >= 0)
    return x;
  else
    return -x;
} //Fim da Função valorAbsoluto(double x)

```

Figura 9.9: Função para cálculo do valor absoluto de um número.

Note que não é possível saber, *a priori*, até onde o fluxo irá, na função *valorAbsoluto*. Caso *x* seja maior ou igual a zero, o fluxo encontrará a primeira instrução *return* e o restante da função será ignorado. Caso *x* seja negativo, a estrutura *if* desvia o fluxo para a segunda instrução *return*. Como uma função termina quando encontra uma instrução *return*, no caso em que esta instrução aparece múltiplas vezes, fica difícil, ou até impossível, determinar, *a priori*, por onde o fluxo irá passar.

Utilizando funções para compor novas funções

O resultado de funções também pode ser armazenado em variáveis ou utilizado para compor expressões mais complexas. Também é possível fazer chamadas de funções dentro de outras funções. As **Figuras 9.10** e **9.11** mostram um exemplo da chamada de uma função dentro de outra função. O problema tratado pelas funções é encontrar o número de combinações possíveis, compostas de *p* elementos escolhidos dentro de um conjunto de *n* elementos. Por exemplo: dado um conjunto *A*, de dez elementos, quantos subconjuntos de três elementos é possível construir utilizando os elementos de *A*? A solução deste problema é dada pela equação da combinação de *n* (10) elementos, tomados *p* (3) a *p* (3). A equação do número de possíveis combinações é a seguinte:

$$c(n, p) = \frac{n!}{p!(n - p)!}$$

Note que, para calcular o número de possíveis combinações, é preciso calcular vários fatoriais. Este é um indicativo de que, se uma função para calcular o fatorial de um número for definida, a solução do problema de encontrar o número de combinações ficará mais fácil. A definição matemática do fatorial de um número é mostrada a seguir:

$$n! = \begin{cases} 1, & \text{sen } n \in \{0, 1\} \\ n(n-1)!, & \text{c. c.} \end{cases}$$

A **Figura 9.10** mostra uma função para calcular o fatorial de um número qualquer x . Note que, se x for igual a zero ou igual a um, a estrutura de repetição não altera o valor de res . Caso x seja maior que um, res é alterada sucessivamente até conter o resultado desejado.

```
public static int fat(int x)
{ //Início da Função
  int res=1;
  for(int i=2; i<=x; i++)
    res*=i;
  return res;
} //Fim da Função fat(int x)
```

Figura 9.10: Função para o cálculo do fatorial.

Uma vez que a função para o cálculo do fatorial já está definida, ela pode ser usada para compor a função do cálculo do número de combinações. A **Figura 9.11** mostra o cálculo do número de combinações de n elementos, tomados p a p , utilizando a função definida na **Figura 9.10**.

```
public static int comb(int n, int p)
{ //Início da Função
  return fat(n)/(fat(p)*fat(n-p));
} //Fim da Função comb(int n, int p)
```

Figura 9.11: Função para o cálculo do número de combinações utilizando a função definida na **Figura 9.10**.

Observe que, uma vez definida a função *fat* para o cálculo do fatorial, basta utilizá-la para compor a função para calcular o número de combinações. Também é possível fazer a função do número de combinações sem o uso de uma função auxiliar, contudo, esta função envolveria três estruturas de repetição: 1. uma para calcular o fatorial de n ; 2. uma para calcular o fatorial de p e 3. uma para calcular o fatorial de $n - p$. Note que isto aumentaria consideravelmente o tamanho (em linhas) da função para o cálculo de combinações.

Também é importante mencionar que expressões aritméticas/lógicas podem ser usadas como parâmetros de função durante sua chamada. Note que a expressão da função definida na **Figura 9.11** apresenta

um $fat(n - p)$. Isto é possível devido ao fato de que as expressões aritméticas/lógicas são resolvidas antes das chamadas das funções. Deste modo, primeiro será computado o resultado da expressão $n - p$ e, depois, serão feitas as chamadas da função fat . Vamos acompanhar passo a passo a execução desta função para $n = 5$ e $p = 3$.

1. A primeira coisa é substituir os valores na expressão: *return fat(n) / (fat(p) * fat(n - p))* é convertido para *return fat(5) / (fat(3) * fat(5 - 3))*.
2. Em seguida, são executadas as expressões aritméticas e lógicas possíveis. Logo, a expressão $5 - 3 = 2$ é resolvida, e a expressão se torna *return fat(5) / (fat(3) * fat(2))*.
3. Para terminar a solução, é necessário executar as chamadas de função. Estas chamadas são avaliadas da esquerda para a direita. Logo, a expressão é convertida para *return 120 / (fat(3) * fat(2))*, depois para *return 120 / (6 * fat(2))* e, por último, *return 120 / (6 * 2)*.
4. Neste ponto, a expressão aritmética é resolvida começando pela parte dentro dos parênteses, sendo convertida para *return 120 / 12*, que é convertida para *return 10*.

Atividade final

Atende aos objetivos 1 e 2

- a) Faça uma função para preencher uma matriz com números informados pelo usuário. Faça também um programa que utilize a função definida para uma matriz 4 X 4.

[illegible]

This image shows a single sheet of white paper with horizontal blue or grey ruling lines. The lines are evenly spaced and run across the width of the page. There are approximately 20 lines visible. The paper has a slight shadow on the right side, suggesting it's resting on a surface.

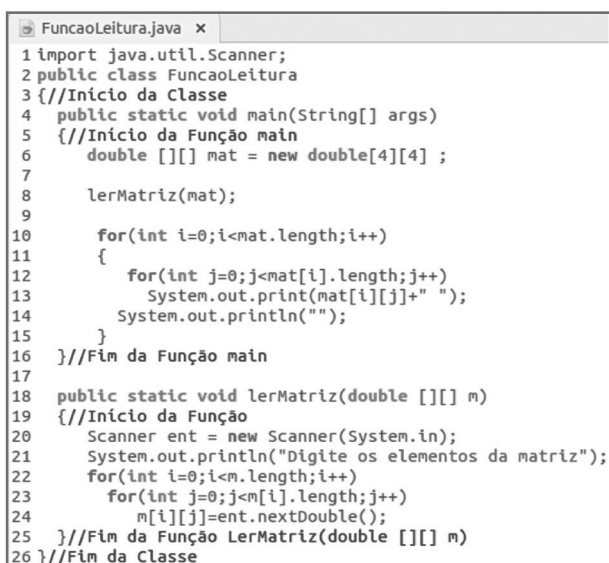
[illegible]

Resposta comentada

a) Vários assuntos podem ser discutidos com esta atividade. Primeiro, deve-se discutir a responsabilidade. Uma vez que temos um programa que utiliza funções, devemos ter claras quais são as responsabilidades de cada função e quais são as responsabilidades do programa. Neste exemplo, a responsabilidade da função é de ler os elementos digitados pelo usuário e armazená-los na matriz. Nenhuma outra atribuição é dada à função pelo enunciado, o que significa que todas as tarefas referentes à leitura devem ser feitas pela função.

As outras responsabilidades são do programa, e, por outras, entende-se alocação da matriz e impressão do resultado. Neste sentido, devemos ter um programa que aloque uma matriz 4 x 4, chame a função para fazer a leitura e, por último, imprima o resultado. Ainda, devemos ter uma função que trata de todas as questões pertinentes à leitura (declaração de variáveis auxiliares, interação com o usuário e a leitura em si).

Outro ponto a ser discutido é o tipo desta função. Como a tarefa de leitura não requer nenhuma resposta específica, esta função pode ser do tipo *void*. A seguir, encontra-se uma possível solução para o problema. Note que, nesta solução, a matriz é declarada e alocada pelo programa, mas seus valores são modificados pela função. Isto é chamado de efeito colateral e será discutido em mais detalhes em momento oportuno.



```

1 import java.util.Scanner;
2 public class FuncaoLeitura
3 { //Início da Classe
4     public static void main(String[] args)
5     { //Início da Função main
6         double [][] mat = new double[4][4] ;
7
8         lerMatriz(mat);
9
10        for(int i=0;i<mat.length;i++)
11        {
12            for(int j=0;j<mat[i].length;j++)
13                System.out.print(mat[i][j]+" ");
14            System.out.println("");
15        }
16    } //Fim da Função main
17
18    public static void lerMatriz(double [][] m)
19    { //Início da Função
20        Scanner ent = new Scanner(System.in);
21        System.out.println("Digite os elementos da matriz");
22        for(int i=0;i<m.length;i++)
23            for(int j=0;j<m[i].length;j++)
24                m[i][j]=ent.nextDouble();
25    } //Fim da Função lerMatriz(double [][] m)
26 } //Fim da Classe

```

Figura 9.12: Possível solução para o problema.

b) Apesar desta questão já ter sido feita na Aula 5, é bom revisita-la para que você possa ver a utilidade de funções neste tipo de situação. A série do seno é composta por uma soma de vários termos e, para cada termo, deve-se calcular um fatorial e uma potenciação. Como uma função para o cálculo do fatorial já foi definida nesta aula, pode-se utilizá-la para criar uma função para o cálculo do seno.

Uma possível solução é mostrada a seguir. O seno de um ângulo é um valor real entre zero e um, logo, é natural que o tipo de retorno da função seja real. Como o objetivo é calcular o seno de um ângulo, *seno* é um nome bastante apropriado para a função. Como parâmetros, é necessário saber o ângulo que será utilizado e o número de termos da série que serão utilizados. Por razões óbvias, o ângulo deve ser um número real e a quantidade de termos um número inteiro. Também são necessárias algumas variáveis auxiliares: *soma*, para armazenar a soma dos termos; *sinal*, para representar a mudança de sinal em cada termo; *i*, para contar a quantidade de termos, e *exp*, para controlar o expoente em cada termo.

```
public static double seno(double x, int limite)
{ //Início da Função
  double soma=0;
  int exp=1,sinal=1;
  for(int i=0;i<limite;i++)
  {
    soma+= sinal*Math.pow(x,exp)/fat(exp); //chamada das funções
    exp+=2;
    sinal*=-1;
  }
  return soma;
} //Fim da Função seno(double x, int limite)
```

Figura 9.13: Possível solução para o problema.

c) Este problema também já foi resolvido nesta disciplina – Aula 7 –, porém, o uso de funções pode dar uma nova perspectiva sobre sua solução. Note que, para calcular a variância de um conjunto, primeiro devemos calcular a média deste conjunto. Portanto, para facilitar a solução do problema, podemos utilizar a função *mediaVetor*, definida nesta aula. Porém, devemos adaptar esta função para que ela receba um conjunto de números reais. Feito isso, vamos analisar a equação da variância. Esta equação pode ser reescrita da seguinte maneira: $\frac{1}{n} \sum_{i=1}^n y_i$, onde $y_i = (x_i - \mu)^2$ e μ é a média dos elementos.

Note que a variância também é a média dos elementos de um segundo conjunto, onde cada elemento deste segundo conjunto é dado como o

quadrado de um elemento do primeiro, subtraído da média. Com este fato em mente, uma possível estratégia é criar um segundo conjunto e utilizar a função que calcula a média de um conjunto para calcular a variância. Uma solução que utiliza esta estratégia é mostrada a seguir. Nesta solução, são mostradas a função para o cálculo da média de um conjunto de valores reais e a função para o cálculo da variância. Para o cálculo da variância, a primeira tarefa é computar a média do conjunto *vet*. Em seguida, é criado um conjunto *aux*, onde cada elemento é definido como a diferença entre um elemento do primeiro conjunto pela média dos elementos elevado ao quadrado. Por último, a variância de *vet* é dada pela média dos elementos de *aux*.

```
public static double mediaVetor(double [] vet)
{ //Início da Função
  double soma=0;
  for(int i=0;i<vet.length;i++)
    soma+=vet[i];
  return soma/vet.length;
} //Fim da Função mediaVetor(int [] vet)

public static double variancia(double[] vet)
{
  double media = mediaVetor(vet);
  double[] aux = new double[vet.length];
  for(int i=0;i<aux.length;i++)
    aux[i]=(vet[i]-media)*(vet[i]-media);

  return mediaVetor(aux);
} //Fim da Função variancia(double[] vet)
```

Figura 9.14: Possível solução para o problema.

Resumo

Nesta aula, você aprendeu a sintaxe de declaração de funções, bem como os mecanismos para utilizá-las. Viu que funções podem retornar resultados que podem ser usados para composição de expressões. Também aprendeu que funções podem ser utilizadas dentro de outras funções, buscando criar soluções para problemas mais complexos.

Definir boas funções e utilizar funções definidas para compor funções mais complexas é uma prática comum entre programadores e é também considerada uma boa prática de programação, uma vez que o uso de funções e procedimentos promove a simplificação e reutilização de código, além de facilitar a manutenção.

Informação sobre a próxima aula

Na próxima aula, você verá como lidar com recursão e com passagem de parâmetros.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T. H. *et al. Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

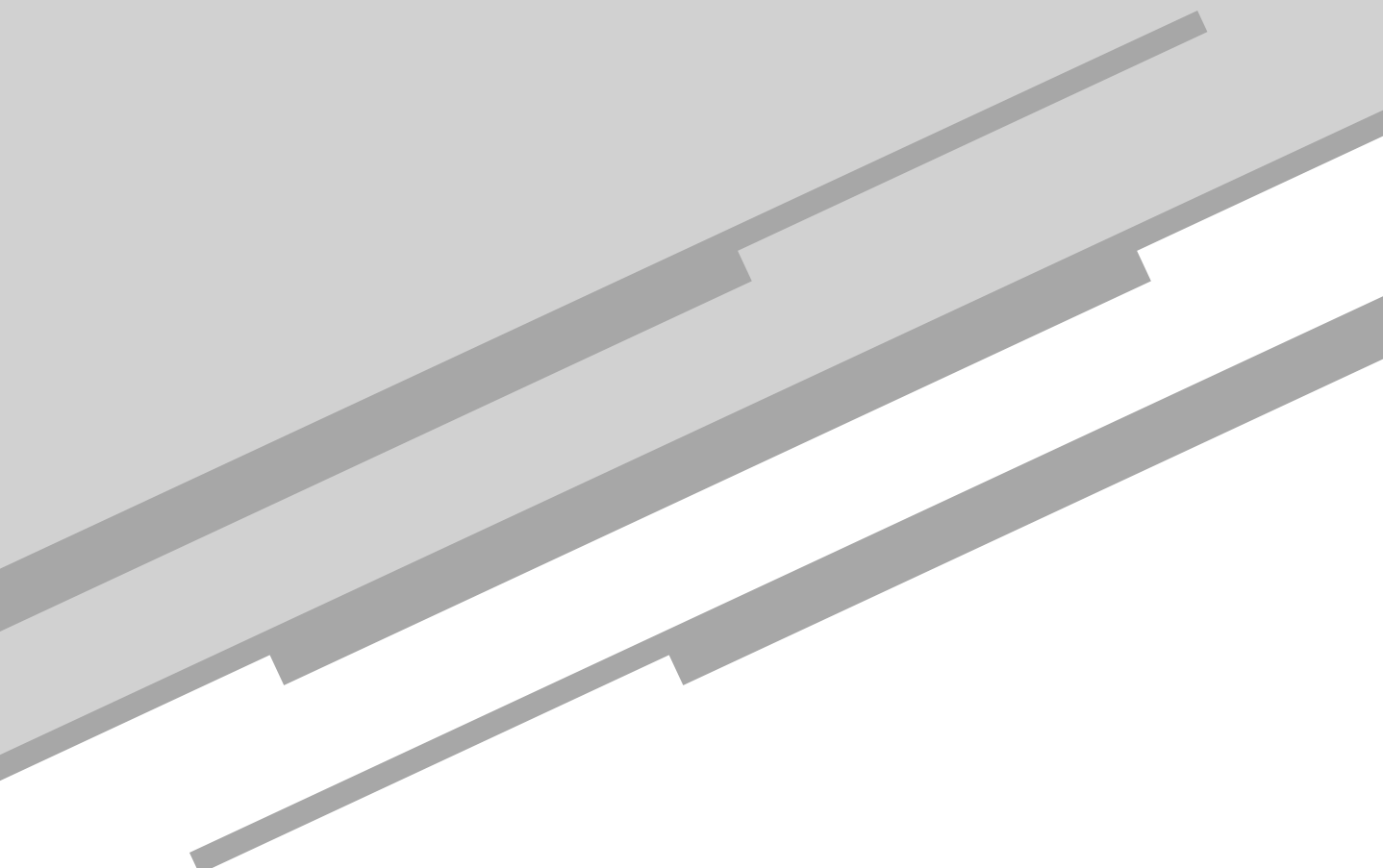
DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. *et al. Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. São Paulo: McGraw-Hill, 1987.

Aula 10

Funções – Parte II



Meta

Expor os conceitos avançados de funções, permitindo a construção de algoritmos mais sofisticados.

Objetivos

Esperamos que, ao final desta aula, você seja capaz de:

1. diferenciar passagem de parâmetros por valor e por referência e entender como Java lida com esta questão;
2. reconhecer e escrever funções recursivas;
3. avaliar se recursão é a estratégia mais adequada para executar uma tarefa qualquer.

Introdução

Como visto em Computação I, as referências são um mecanismo importante e vital para algumas tarefas do nosso dia a dia, bem como na própria computação. Imagine a dificuldade que seria enviar uma encomenda sem o sistema de referências usado atualmente (endereços).

Nesta aula, você verá como os conceitos de referência e recursão são tratados em Java. Estes conceitos permitem construir funções com múltiplos retornos e possibilitam a construção de códigos mais simples e elegantes, sendo, portanto, conceitos fundamentais para a programação.

Referências

Até o momento, você tem usado passagem de parâmetros sem saber se está fazendo isso por valor ou por referência. A dica é que, em Java, todas as passagens de parâmetros são feitas por valor, embora seja possível alterar os valores de alguns tipos de parâmetros.

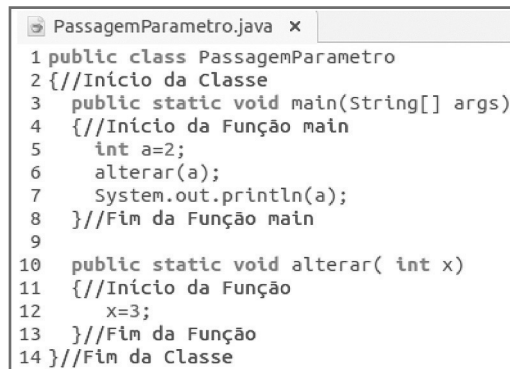
Antes de prosseguir, vamos relembrar o que são as passagens de parâmetro por valor e por referência.

A **passagem por valor** cria cópias dos parâmetros reais e passa as cópias para as funções, que não têm acesso aos dados originais. Isto impossibilita a alteração dos parâmetros originais e, portanto, impede que se possa fazer uma função que altere os elementos de um vetor passado como parâmetro, por exemplo. Além disso, deve-se levar em conta que, apesar de ser feita de modo transparente para os programadores, a criação de cópias demanda tanto memória quanto tempo, fazendo com que programas que fazem o uso desnecessário de parâmetros passados por valor consumam mais memória e demorem mais para concluir sua execução.

Já na **passagem por referência**, são criadas referências para os elementos originais. A ideia é similar a de informar um endereço para entrega de um produto. Quando você faz uma compra pela Internet, você simplesmente preenche um endereço de entrega (referência) para que a sua compra chegue até sua casa. Note que apesar de o endereço não ser sua casa (o endereço é só uma informação sobre a localização da casa), ele é suficiente para que o produto possa ser entregue.

Em Java, as passagens são feitas por valor, o que significa que são feitas cópias e estas são passadas para as funções. Na **Figura 10.1**, há uma função que faz a alteração do valor de uma variável passada como

parâmetro. Como a passagem é feita por valor nada ocorre com a variável original.



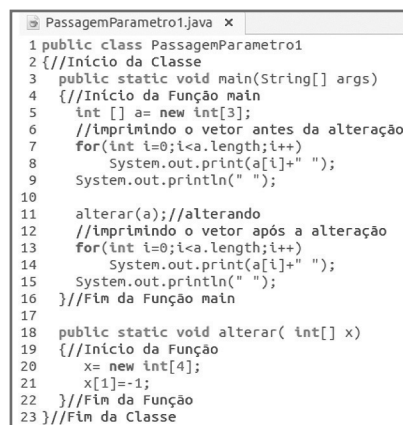
```

1 public class PassagemParametro
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         int a=2;
6         alterar(a);
7         System.out.println(a);
8     } //Fim da Função main
9
10    public static void alterar( int x)
11    { //Início da Função
12        x=3;
13    } //Fim da Função
14 } //Fim da Classe

```

Figura 10.1: Exemplo de passagem por valor sem alteração. Ao executar o programa, o que será impresso é o valor original da variável *a*, apesar do comando de atribuição executado dentro da função *alterar*.

De modo similar, ao executar o programa da **Figura 10.2**, o vetor impresso no início será o mesmo impresso no fim.



```

1 public class PassagemParametro1
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         int [] a= new int[3];
6         //imprimindo o vetor antes da alteração
7         for(int i=0;i<a.length;i++)
8             System.out.print(a[i]+" ");
9         System.out.println(" ");
10
11        alterar(a); //alterando
12        //imprimindo o vetor após a alteração
13        for(int i=0;i<a.length;i++)
14            System.out.print(a[i]+" ");
15        System.out.println(" ");
16    } //Fim da Função main
17
18    public static void alterar( int[] x)
19    { //Início da Função
20        x= new int[4];
21        x[1]=-1;
22    } //Fim da Função
23 } //Fim da Classe

```

Figura 10.2: Exemplo de passagem por valor usando nova alocação. Apesar da função *alterar* alocar um novo vetor, a passagem por valor garante que o vetor original não é alterado.

Agora observe a **Figura 10.3**, na qual é passado um vetor como parâmetro de uma função – assim como na **Figura 10.2** –, porém o resultado é bem diferente do o que é esperado para uma passagem de parâmetro por valor.

O vetor impresso no fim é diferente do vetor impresso no início.

```

PassagemParametro2.java x
1 public class PassagemParametro2
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         int [] a= new int[3];
6         //imprimindo o vetor antes da alteração
7         for(int i=0;i<a.length;i++)
8             System.out.print(a[i]+" ");
9         System.out.println(" ");
10
11         alterar(a); //alterando
12         //imprimindo o vetor após a alteração
13         for(int i=0;i<a.length;i++)
14             System.out.print(a[i]+" ");
15         System.out.println(" ");
16     } //Fim da Função main
17
18     public static void alterar( int[] x)
19     { //Início da Função
20         x[1]=-1;
21     } //Fim da Função
22 } //Fim da Classe

```

Figura 10.3: Exemplo de passagem por valor que faz alteração. Neste exemplo, o vetor original é alterado.

Este tipo de alteração é uma das causas para vários programadores se confundirem sobre as passagens de parâmetro em Java. Apesar de a alteração ter sido de fato feita, houve uma passagem de parâmetro por valor. Porém, o que foi passado por valor foi uma referência.

Observe a linha 5. Nesta linha, são criados uma referência (*a*) e um vetor (*new int[3]*) e o vetor é associado à referência. Ao passarmos *a* para a função, passamos a referência, e não o vetor.

Exemplificando, suponha que o vetor criado é armazenado na posição 175 da memória. Como sabemos, a referência armazena o valor 175, que é o endereço onde o vetor está armazenado. Ao passarmos a referência *a* para a função, cria-se uma referência *x*, que é uma cópia da referência *a*. Logo, *x* também contém o valor 175. Isto implica que o vetor em si não é copiado, apenas sua referência, e, ao se fazer alterações dentro da função, altera-se o objeto apontado pela referência (neste caso o vetor contido na posição 175), e não na referência. Assim, o valor contido em *a* depois da execução da função continua o mesmo (175), porém o vetor foi alterado.

Já na **Figura 10.2**, dentro da função, ocorre a criação de um novo vetor e este novo vetor é associado à referência *x*. Note que a posição do novo vetor não pode ser a mesma do vetor referenciado por *a* (a memória utilizada pelo vetor referenciado por *a* não pode ser usada enquanto este vetor não for desalocado). Logo, as alterações são feitas em um vetor diferente, fazendo com que o vetor apontado por *a* fique intacto.

Na **Figura 10.1** não há criação de referências, uma vez que, ao usarmos tipos primitivos (como *double*, *int* e *boolean*, por exemplo), são criadas variáveis convencionais. As referências são usadas apenas na criação de objetos (*String*, *Scanner*, vetores e matrizes, por exemplo).

Note que a criação de referências em Java é feita de maneira transparente, visto que você já tem usado referências para objetos do tipo *String* e *Scanner* sem que isso tenha implicado o uso de sintaxes diferenciadas. Em C++, por exemplo, existe distinção na sintaxe de criação de referências e variáveis. As variáveis são criadas exatamente como em Java. As referências precisam ser marcadas com a colocação de um & (“e” comercial) em seus tipos. Note que, com o uso de referências, é possível alterar os parâmetros de entrada, desde que não sejam atribuídos outros objetos às referências durante a execução da função.



A capacidade de alterar parâmetros de entrada em funções tem o nome de *efeito colateral* e, como o nome sugere, seu uso deve ser feito com cautela.

Programas que fazem uso excessivo deste recurso são, na maioria dos casos, difíceis de entender e, portanto, quando ocorrem erros, difíceis de corrigir. Na verdade, a própria detecção do local onde o erro ocorre é muito difícil em alguns casos.

Atividade 1

Atende ao objetivo 1

Faça uma função que receba um vetor de números reais e um número inteiro. A função deve ordenar os elementos do vetor. Caso o valor do número inteiro passado for maior que zero, a ordem deve ser crescente. Caso contrário, a ordem deve ser decrescente.

Resposta comentada

Como os parâmetros já foram determinados pelo enunciado, a primeira coisa a fazer é determinar o tipo desta função. Como o objetivo é ordenar os elementos de um vetor, que é passado como parâmetro, nenhuma informação adicional precisa ser retornada. Logo esta função pode ser do tipo *void*. Uma possível solução é mostrada a seguir:

A solução pode ser dividida em duas partes: uma para ordenar em ordem crescente e outra para ordenar em ordem decrescente. Note que a estratégia quanto ao que foi feito em cada parte difere um pouco da estratégia mostrada na Aula 7. A solução apresentada anteriormente continha uma verificação a mais para efeitos didáticos. A troca de posição ocorria somente se *pos* fosse diferente de *i*. Como você pode observar, esta comparação, apesar de coerente, não é de fato necessária e removê-la não implica erros no algoritmo. Quando esta comparação é removida, o que ocorre é que, caso *i* e *pos* sejam iguais, há uma troca entre os elementos de mesma posição, ou seja, o número contido na posição *i* é substituído por um número com o mesmo valor.

```
public static void ordenar(double [] vetor, int orden)
{
    if(orden>0)//verifica se deve ser crescente ou decrescente
    {
        for(int i=0;i<vetor.length-1;i++)//ordenação crescente
        {
            double menor= vetor[i];
            int pos = i;
            for(int j=i+1;j<vetor.length;j++)//busca elemento de prioridade
            {
                if(vetor[j]<menor)//toda vez que um novo elemento de maior
                { //prioridade é encontrado substitui-se o antigo
                    menor=vetor[j];
                    pos=j;
                }
            }
            double aux= vetor[i];//trocando o elemento atual com o de
            vetor[i]=menor; //prioridade
            vetor[pos]=aux;
        }
    }
    else
    {
        for(int i=0;i<vetor.length-1;i++)//ordenação decrescente
        {
            double maior= vetor[i];
            int pos = i;
            for(int j=i+1;j<vetor.length;j++)//busca elemento de prioridade
            {
                if(vetor[j]>maior)//toda vez que um novo elemento de maior
                { //prioridade é encontrado substitui-se o antigo
                    maior=vetor[j];
                    pos=j;
                }
            }
            double aux= vetor[i];//trocando o elemento atual com o de
            vetor[i]=maior; //prioridade
            vetor[pos]=aux;
        }
    }
}
//public static void ordenar(double [] vetor, int orden)
```

Figura 10.4: A solução pode ser dividida em duas partes: uma para ordenar em ordem crescente e outra para ordenar em ordem decrescente.

Recursão

Você certamente já utilizou alguma recursão ao longo dos seus estudos. A recursão é uma técnica geral de definir algo (funções, estruturas, atividades) em função de instâncias menores deste algo. Na matemática, esta técnica é muito utilizada na definição de funções e estruturas matemáticas. Na computação, é frequentemente utilizada na definição de subprogramas (funções e procedimentos).

Uma definição matemática recursiva muito conhecida é a definição do fatorial, que é dada a seguir:

$$n! = \begin{cases} 1, & \text{se } n \in \{0,1\} \\ n(n-1)!, & \text{c. c.} \end{cases}$$

Note que, pela definição, o problema é facilmente resolvido quando n vale 0 ou 1. Para qualquer outro valor de n , multiplica-se n pelo resultado da mesma função fatorial aplicada a $n - 1$. Ou seja, quando usamos a função fatorial recursivamente para resolver o fatorial de $n - 1$.

Toda recursão tem, pelo menos, duas partes. Uma é chamada de caso base e a outra de caso recursivo. Em alguns casos, podem existir mais de um caso base e mais de um caso recursivo, mas há, pelo menos, um caso base e um caso recursivo em qualquer definição recursiva.

Os casos base são os casos em que o problema é facilmente resolvido. No caso do fatorial, este caso ocorre quando n é igual a zero ou igual a um. O caso recursivo é aquele em que a própria função é usada, recursivamente, pelo menos uma vez. Normalmente, o caso recursivo utiliza a função para resolver uma instância menor do problema e utiliza o resultado obtido nesta instância menor para resolver o problema original. No exemplo do fatorial, este caso ocorre quando n é maior que um, que é quando o resultado do fatorial de $n-1$ é calculado recursivamente e utilizado para compor uma multiplicação que resulta na solução do problema original.

A recursão provê mecanismos simples e elegantes para a solução de diversos problemas, e por isso é considerada uma técnica fundamental dentro da teoria da computação. Para desenvolver uma solução recursiva, o primeiro passo é identificar os casos base e os casos recursivos. Uma vez identificados os casos, deve-se construir os algoritmos recursivos que implementem a solução.



A recursão em diversas linguagens

Em algumas linguagens, como C++ e Java, funções recursivas são facilmente implementadas. Entretanto, o uso de recursão em algumas linguagens, como Fortran, não é trivial. Existem inclusive

algumas linguagens que são naturalmente recursivas, como Haskell. Portanto, dependendo da linguagem utilizada no projeto em que você estiver trabalhando, utilizar funções recursivas pode ser uma boa opção.

Uma possível implementação recursiva para o fatorial é dada pela **Figura 10.5**. Note que a função se aproxima bem mais da definição matemática do que as versões feitas até o momento.

```
public static int fatorial(int n)
{
    //Início da Função
    if(n==1 || n==0 )
        return 1;
    else
        return n * fatorial(n-1);
} //Fim da Função int fatorial(int n)
```

Figura 10.5: Função para o cálculo do fatorial recursiva.

As chamadas de uma função recursiva são resolvidas em uma estratégia de empilhamento, ou seja, a última função a ser chamada é a primeira a ser resolvida. Para exemplificar o mecanismo de chamadas para uma função recursiva, será utilizada a função fatorial definida na **Figura 10.6** aplicada a 5. Note que 5 não se encaixa no caso base, o que faz com que $fatorial(5)$ seja analisado como $5 * fatorial(4)$. Perceba que, para calcular o fatorial de 5, deve-se calcular o fatorial de 4, o que faz com que o fatorial de 4 deva ser resolvido primeiro para que o fatorial de 5 possa ser calculado. Este $fatorial(4)$ resulta em uma nova chamada da função fatorial onde o parâmetro também não se encaixa no caso base, o que resultará em $4 * fatorial(3)$. O $fatorial(3)$ resulta em $3 * fatorial(2)$, que por sua vez resulta em $2 * fatorial(1)$. Neste ponto, a recursão encontra um caso base e retorna um resultado que é retornado para o ponto onde a chamada recursiva foi feita. Neste caso $2 * fatorial(1)$ é avaliado como $2 * 1$, que resulta em 2. Este resultado é retornado para o ponto onde fatorial de 2 foi chamado; neste caso, $3 * fatorial(2)$ é avaliado como $3 * 2$, que resulta em 6. Este processo de retornos sucessivos continua até que todas as chamadas recursivas tenham sido resolvidas e o resultado final do fatorial seja calculado.

Na **Figura 10.6**, (a) mostra o processo de chamadas recursivas para a função fatorial. Já (b) mostra o processo de retorno na pilha de recursão.

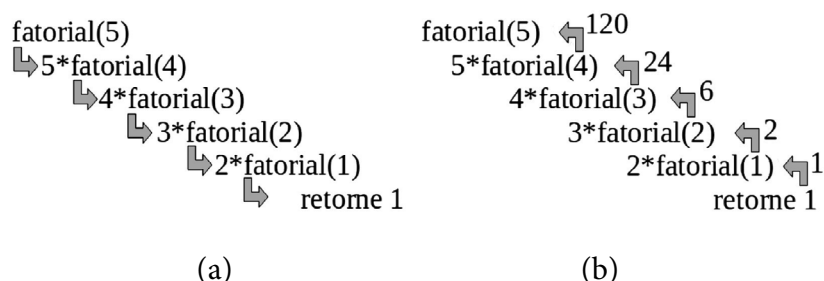


Figura 10.6: Pilha de execução: chamadas (a) e retornos (b).

Outro exemplo clássico de uso de recursão é o cálculo da série de Fibonacci. A definição matemática dos termos da série é dada a seguir.

$$fib(n) = \begin{cases} 1, & \text{se } n \in \{1, 2\} \\ fib(n-1) + fib(n-2), & \text{c. c.} \end{cases}$$



Segundo a história, no século XIII, o matemático Leonardo Pisa, também conhecido como Fibonacci, propôs uma sequência numérica para explicar o crescimento de uma população de coelhos. Esta sequência ficou conhecida como série de Fibonacci. Desde então, muitos matemáticos se dedicaram ao estudo desta série e várias aplicações práticas, além do controle populacional, foram encontradas.

Função iterativa

Função que não usa recursão. Algoritmo e/ou programa de computador que realiza suas tarefas através da repetição de uma ou mais ações. Cada ciclo de repetição das ações é chamado de iteração.

Note que, assim como o fatorial, a série de Fibonacci tem um caso base e um caso recursivo, porém no caso recursivo existem duas chamadas à função *fib*: uma usando $n - 1$ como parâmetro e outra usando $n - 2$.

Na **Figura 10.7**, é mostrada uma possível implementação de uma função para o cálculo da série de Fibonacci (a), bem como uma versão **iterativa** (que usa estrutura de repetição) vista anteriormente (b).

Note que, além de ser menor, a versão recursiva se aproxima mais da definição matemática e também é mais simples que a versão iterativa.

```
public static int fibRec(int n)
{
    //Início da Função
    if(n==2 || n==1 )
        return 1;
    else
        return fibRec(n-1) + fibRec(n-2);
}
//Fim da Função fibRec(int n)
```

(a)

```
public static int fibIter(int n)
{
    //Início da Função
    int fib1=1, fib2=1, res=1;
    for(int i=3; i<=n; i++)
    {
        res=fib1 + fib2;
        fib2=fib1;
        fib1=res;
    }
    return res;
}
//Fim da Função fibIter(int n)
```

(b)

Figura 10.7: Funções para o cálculo da Série de Fibonacci: recursiva (a) e iterativa (b).

Apesar de a recursão oferecer uma maneira simples e prática para solucionar problemas, ela costuma ser menos eficiente, principalmente pela questão do retrabalho. A versão recursiva para o cálculo da série de Fibonacci é um bom exemplo. A **Figura 10.8** mostra todas as chamadas feitas para o cálculo do quinto termo da série de Fibonacci. Note que, para concluir o cálculo de *fib(5)*, é necessário resolver duas vezes *fib(3)* e três vezes *fib(2)*.

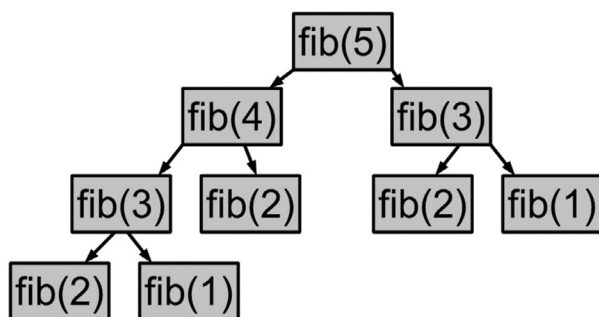


Figura 10.8: Análise das chamadas recursivas para o cálculo da série de Fibonacci.

É importante mencionar que, mesmo em casos onde não há retrabalho, versões recursivas de funções ainda são mais lentas quando implementadas em computadores. Isso, pelo fato de chamadas de função serem avaliadas mais lentamente do que operações matemáticas convencionais.

A **Figura 10.9** mostra um programa que faz uma comparação simples entre os tempos de execução de duas funções. Este programa faz 5 mil chamadas de cada uma das funções e mostra o volume de tempo consumido pelas chamadas recursivas e pelas chamadas iterativas. Os tempos são mostrados em milissegundos. A execução deste programa em um computador com processador Intel Core-I7® com 3.4 GHz por core, 64-bits, com 16 GB de memória principal, rodando sistema operacional Linux Kernel 3.11, mostra que as chamadas recursivas demoram o dobro do tempo gasto pelas chamadas iterativas.



```

1 public class FuncaoFatorialRecursivo
2 { //Início da Classe
3     public static void main(String[] args)
4     { //Início da Função main
5         double t1 = System.currentTimeMillis(); //tempo no início
6         for(int i=0; i<5000; i++)
7             fatorial(5); //função recursiva
8
9         double t2 = System.currentTimeMillis(); //tempo no meio
10
11        for(int i=0; i<5000; i++)
12            fat(5); //função iterativa
13
14        double t3 = System.currentTimeMillis(); //tempo no fim
15
16        System.out.println("Tempo recursivo: "+(t2-t1));
17        System.out.println("Tempo iterativo: "+(t3-t2));
18    } //Fim da Função main
19
20    public static int fatorial(int n)
21    { //Início da Função
22        if(n==1 || n==0)
23            return 1;
24        else
25            return n * fatorial(n-1);
26    } //Fim da Função int fatorial(int n)
27
28    public static int fat(int n)
29    { //Início da Função
30        int result=1;
31        for(int i=2; i<=n; i++)
32            result*=i;
33        return result;
34    } //Fim da Função int fat(int n)
35
36
37 } //Fim da Classe

```

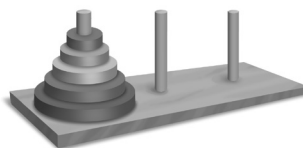
Figura 10.9: Programa que compara os tempos de execução de uma função iterativa e de uma função recursiva.

Existem teoremas que afirmam que todo algoritmo recursivo pode ser descrito de forma iterativa e vice-versa. Neste sentido, o programador deve avaliar o impacto do uso de recursão nos seus programas. Caso este impacto seja relativamente pequeno em vista da simplicidade proporcionada na solução, o uso de recursão pode ser uma boa opção.

Atividade 2

Atende aos objetivos 2 e 3

A torre de Hanói é um brinquedo muito conhecido. Existem três pinos e uma série de discos de diâmetros diferentes. O objetivo do jogo é mudar todos os discos de um pino para outro, podendo usar o



terceiro pino como auxiliar, sem que um disco de diâmetro maior seja empilhado sobre um disco de diâmetro menor, embora discos de diâmetro menor possam ser empilhados sobre discos de diâmetro maior.

Faça uma função que, dado um número n de discos a serem transferidos, calcule quantos movimentos são necessários para completar o jogo, em que um movimento compreende tirar um disco de um pino e colocar em outro. Sabe-se que o número de movimentos necessários é dado por uma função $h(n)$, definida como:

$$h(n) = \begin{cases} 1, & \text{se } n = 1 \\ 1 + 2 h(n-1), & \text{c. c.} \end{cases}$$

Resposta comentada

Seguindo a definição recursiva e os conteúdos passados, esta questão tem implementação simples. Uma possível solução é apresentada a seguir:

```

public static int numHanoi(int n)
{
    if(n==1)
        return 1;
    else
        return 1+ 2* numHanoi(n-1);
} //Fim da função int numHanoi(int n)

```

Figura 10.10: Possível solução com uso de recursão.

Atividade 3

Atende ao objetivo 1

Faça uma função que receba como parâmetro uma matriz de números reais e altere os elementos desta matriz, zerando todos os elementos, exceto os da diagonal principal. A soma dos números que foram apagados também deve ser calculada.

Resposta comentada

Esta questão envolve atividades frequentes relacionadas com matrizes, alteração e somas parciais:

```

public static double somarForaDiagonalApagarElementos(double[][] mat)
{
    double result=0;
    for(int i=0;i<mat.length;i++)
    {
        for(int j=0;j<mat[i].length;j++)
        {
            if(i!=j)
            {
                result+= mat[i][j];
                mat[i][j]=0;
            }
        }
    }
    return result;
}

```

Figura 10.11: Possível solução para o problema.

Conclusão

Nesta aula, foi mostrado como funciona a passagem de parâmetros em Java. Todos os parâmetros são passados por valor, contudo, não há criação de cópias desnecessárias, visto que os vetores e matrizes (e outros objetos) têm suas referências passadas por valor, e não pelos dados em si. Neste caso, não há perda de eficiência significativa, pois o mecanismo utilizado evita cópias desnecessárias, além de permitir a alteração de objetos que têm suas referências passadas como parâmetros.

Quanto à recursão, o único inconveniente para seu uso é o desempenho: funções recursivas são mais lentas que funções iterativas. Se você não está preocupado com a eficiência dos seus programas, o uso de recursão pode trazer grande simplificação destes. Agora, se a eficiência é um fator crítico, a recursão não é recomendada. Para esta disciplina, a eficiência de um programa não é um dos critérios de avaliação. O objetivo aqui é fazer com que você desenvolva sua lógica e consiga aplicá-la na construção de programas. Portanto, caso você proponha uma solução recursiva para um problema, e esta solução esteja correta, ela será considerada tão boa quanto uma solução iterativa.

Ao longo da disciplina, você viu várias técnicas de construção de programas, utilizando ferramentas lógicas e matemáticas. Os recursos apresentados possuem ferramentas equivalentes na maioria das linguagens de programação imperativas, o que lhe torna apto a propor soluções

computacionais para diversos problemas. Estas linguagens possuem mecanismos de desvio condicional, repetição, uso de arquivos, criação de funções e variáveis, de modo que, se você precisar utilizar uma linguagem diferente da que foi utilizada, tudo o que você tem a fazer é entender como estes conceitos funcionam na nova linguagem.

Agora é hora de começar a utilizar o seu conhecimento na construção de programas de computador para resolver problemas de engenharia, física e matemática. Para isso, você deve cursar a disciplina Métodos Numéricos. Outras disciplinas que podem ajudar na elaboração de programas mais sofisticados e eficientes são as disciplinas de Estrutura de Dados e Programação Orientada a Objetos, normalmente oferecidas pelos cursos de Sistema de Informação e Ciência da Computação. Caso você tenha interesse nas áreas de computação, simulação ou pesquisa operacional, cursar estas disciplinas como optativas pode ser uma boa opção. Boa sorte!

===== **Atividade final** =====

Atende aos objetivos 1, 2 e 3

O máximo divisor comum (MDC) entre dois números inteiros a e b , denotado por $\text{mdc}(a,b)$, pode ser obtido pela relação descrita abaixo. Faça uma função que calcule o MDC recursivamente, de acordo com esta definição. A função construída não deve fazer efeitos colaterais nos parâmetros.

$$\text{mdc}(a,b) = \begin{cases} a, & \text{se } b = 0 \\ \text{mdc}(b, a \% b), & \text{c. c.} \end{cases}$$

Resposta comentada

Uma vez estabelecido o padrão da recursão, a solução deste problema é relativamente simples. Uma possível solução é dada a seguir. Você fez uma versão iterativa (sem recursão) desta questão na disciplina Computação I. Se você tentar implementar a versão iterativa em Java, verá que a solução recursiva é bem mais simples. A questão dos efeitos colaterais poderia ser resolvida usando tipos primitivos (*int*, *double*, etc.), pois estes são passados por valor e, portanto, seus valores originais não podem ser alterados. Note que a solução apresentada utiliza esta estratégia:

```
public static int mdc(int a, int b)
{
    if(b==0)
        return a;
    else
        return mdc(b,a%b);
} //Fim da função int mdc(int a, int b)
```

Figura 10.12: Possível solução para a questão.

Resumo

Nesta aula, você aprendeu como funciona a passagem de parâmetros em Java e que esta passagem é sempre feita por valor. Também aprendeu a lidar com a técnica de recursão, que utiliza uma função dentro dela mesma, tornando a definição de algumas funções mais simples.

Referências

ASCENCIO, A. F. G.; CAMPOS, E. A. V. *Fundamentos da programação de computadores*. São Paulo: Pearson, 2012.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3. ed. Cambridge: MIT Press, 2009.

DEITEL, P.; DEITEL, H. *Java: How to Program*. 9th ed. Boston: Prentice Hall, 2012.

FARRER, H. et al. *Programação estruturada de computadores*. 2. ed. Rio de Janeiro: Guanabara, 1989.

STEINBRUCH, A.; WINTERLE, P. *Geometria analítica*. 2. ed. São Paulo: McGraw-Hill, 1987.

