e-Tec Brasil
Escola Técnica Aberta do Brasil

Técnicas de Programação

Antonio Luiz Santana

Curso Técnico em Informática




INSTITUTO FEDERAL
ESPÍRITO SANTO

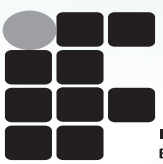
UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Ministério da Educação

e-Tec Brasil
Escola Técnica Aberta do Brasil

Técnicas de Programação

Antonio Luiz Santana



INSTITUTO FEDERAL
ESPÍRITO SANTO

Colatina-ES
2011

Presidência da República Federativa do Brasil

Ministério da Educação

Secretaria de Educação a Distância

Este Caderno foi elaborado em parceria entre o Instituto Federal do Espírito Santo e a Universidade Federal de Santa Catarina para o Sistema Escola Técnica Aberta do Brasil – e-Tec Brasil.

Equipe de Elaboração
Instituto Federal do Espírito Santo – IFES

Coordenação Institucional
Guilherme Augusto de Morais Pinto/IFES
João Henrique Caminhas Ferreira/IFES

Coordenação Curso
Allan Francisco Forzza Amaral/IFES

Professor-autor
Antonio Luiz Santana/IFES

Comissão de Acompanhamento e Validação
Universidade Federal de Santa Catarina – UFSC

Coordenação Institucional
Araci Hack Catapan/UFSC

Coordenação do Projeto
Silvia Modesto Nassar/UFSC

Coordenação de Design Instrucional
Beatriz Helena Dal Molin/UNIOESTE e UFSC

Coordenação de Design Gráfico
André Rodrigues/UFSC

Design Instrucional
Gustavo Pereira Mateus/UFSC

Web Master
Rafaela Lunardi Comarella/UFSC

Web Design
Beatriz Wilges/UFSC
Mônica Nassar Machuca/UFSC

Diagramação
André Rodrigues da Silva/UFSC
Bárbara Zardo/UFSC
Caroline Ferreira da Silva/UFSC
Juliana Tonietto/UFSC
Nathalia Takeuchi/UFSC

Revisão
Júlio César Ramos/UFSC

Projeto Gráfico
e-Tec/MEC

S232t Santana, Antonio Luiz

Técnicas de programação : Curso Técnico em Informática / Antonio Luiz Santana. – Colatina: Ifes, 2011.
114 p. : il.
Inclui Bibliografia

ISBN: 978-85-62934-01-8
1. Java (Linguagem de programação de computador. 2. Informática. I. Instituto Federal do Espírito Santo. II. Título.

CDD: 005.133

Apresentação e-Tec Brasil

Prezado estudante,

Bem-vindo ao e-Tec Brasil!

Você faz parte de uma rede nacional pública de ensino, a Escola Técnica Aberta do Brasil, instituída pelo Decreto nº 6.301, de 12 de dezembro 2007, com o objetivo de democratizar o acesso ao ensino técnico público, na modalidade a distância. O programa é resultado de uma parceria entre o Ministério da Educação, por meio das Secretarias de Educação a Distancia (SEED) e de Educação Profissional e Tecnológica (SETEC), as universidades e escolas técnicas estaduais e federais.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

O e-Tec Brasil leva os cursos técnicos a locais distantes das instituições de ensino e para a periferia das grandes cidades, incentivando os jovens a concluir o ensino médio. Os cursos são ofertados pelas instituições públicas de ensino e o atendimento ao estudante é realizado em escolas-polo integrantes das redes públicas municipais e estaduais.

O Ministério da Educação, as instituições públicas de ensino técnico, seus servidores técnicos e professores acreditam que uma educação profissional qualificada – integradora do ensino médio e educação técnica, – é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação
Janeiro de 2010

Nosso contato
etecbrasil@mec.gov.br

Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.

Atenção: indica pontos de maior relevância no texto.

Saiba mais: oferece novas informações que enriquecem o assunto ou "curiosidades" e notícias recentes relacionadas ao tema estudado.

Glossário: indica a definição de um termo, palavra ou expressão utilizada no texto.

Mídias integradas: sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.

Atividades de aprendizagem: apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

Sumário

Palavra do professor-autor

Olá caro estudante!

Parabéns, caro estudante! Você está iniciando mais uma etapa do Curso Técnico em Informática a distância. A equipe instrucional elaborou todo o material necessário ao suporte para o seu aprendizado. Neste formato, a disciplina Técnicas de Programação foi elaborada pensando numa leitura rápida e dinâmica, abordando o centro de cada conteúdo, explanado em aulas bem objetivas. Como já é do seu conhecimento, estudar a distância é uma tarefa que envolve sua aplicação na resolução dos exercícios, contando com todo amparo da equipe que irá apoiá-lo no processo de ensino-aprendizagem. Para que isso ocorra de forma efetiva, faz-se necessário separar um tempo para estudar o material e fazer as leituras complementares indicadas no caderno. Esperamos que você utilize todos os recursos do ambiente disponíveis para dar andamento aos estudos e avançar pelos módulos.

Um grande abraço!

Prof. Antonio Luiz Santana

Apresentação da disciplina

Nesta disciplina vamos estudar cinco tópicos que precisamos utilizar com muita frequência: conceitos e aplicações de tipos de dados; técnicas de modularização; passagem de parâmetros e recursividade; ambientes e técnicas de desenvolvimento de aplicações; e estruturas de dados e seus algoritmos. Para este nosso estudo, vamos adotar Java como linguagem para desenvolver aplicações. Para a digitação do código fonte das classes Java, a única ferramenta necessária é o bloco de notas do Windows; entretanto, qualquer editor de textos disponível na máquina do leitor pode ser utilizado.

Nas três primeiras aulas, abordaremos os conceitos iniciais de Java e um estudo de variáveis e estruturas básicas de programação. Nas três aulas subseqüentes, apresentaremos os tipos de estrutura de dados e suas aplicações em Java. Em seguida, abordaremos assuntos específicos sobre modularização e recursividade.

Como em qualquer outra linguagem, há muitas opções no mercado e diversas maneiras de desenvolver aplicações em Java. Existe uma infinidade de ferramentas que podem deixar o desenvolvedor com dúvidas para selecionar o ambiente de trabalho. No momento, as ferramentas que mais se destacam são Eclipse e Netbeans. Dessa forma, ao final desta disciplina você estará capacitado a utilizar esses ambientes de desenvolvimento em situações comuns nas empresas, identificando o que melhor se adapta à solução de um determinado problema.

Mesmo que você já tenha estudado alguns desses programas, não deixe de ler o conteúdo semanal da matéria e resolver as atividades propostas. Participe também das discussões com os tutores e demais colegas de curso; você sempre aprenderá uma nova forma de resolver determinado problema. Organize seu tempo reservando um horário todos os dias para os estudos, para que as atividades não acumulem.

E lembre-se: a melhor forma de aprender é praticando! Todo dia descobrimos um novo recurso ou uma nova utilização para esses ambientes de desenvolvimento.

Um grande abraço!

Projeto instrucional

Disciplina: Técnicas de Programação (carga horária: 90 horas).

Ementa: Conceitos e aplicações de tipos de dados. Técnicas de modularização, passagem de parâmetros e recursividade. Ambientes e técnicas de desenvolvimento de aplicações. Estruturas de dados e seus algoritmos. Parte 1: Conceitos e aplicações de tipos de dados. Técnicas de modularização, passagem de parâmetros e recursividade. Ambientes e técnicas de desenvolvimento de aplicações. Parte 2: Estruturas de dados e seus algoritmos.

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
1. Visão geral (Introdução à linguagem Java e plataforma)	- Descrever as principais características da linguagem. - Descrever os procedimentos necessários para o desenvolvimento de uma aplicação Java. - Fornecer ao aluno o primeiro contato com a linguagem Java.	Caderno e Ambiente Virtual de Ensino- Aprendizagem. www.cead.ifes.edu.br	10
2. Aspectos fundamentais sobre Java	- Demonstrar a declaração de dados. - Verificar os conversores de tipo em Java.	Caderno e Ambiente Virtual de Ensino- Aprendizagem. www.cead.ifes.edu.br	10
3. Estruturas condicionais e estruturas de controle	- Fornecer conhecimentos para utilizar corretamente as estruturas condicionais. - Verificar as diferentes estruturas de repetição. - Verificar aplicações práticas.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. www.cead.ifes.edu.br	10
4. Funções matemáticas e de *string*	- Demonstrar as principais funções matemáticas em Java. - Demonstrar os principais métodos para manipulação de *strings* em Java. - Mostrar as técnicas de localização de caracteres em *strings*.	Caderno e Ambiente Virtual de Ensino- Aprendizagem. www.cead.ifes.edu.br	10

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
5. Criando funções	- Identificar os principais tipos de métodos em Java. - Introduzir o conceito de modularidade. - Mostrar as técnicas de criação de métodos em Java.	Caderno e Ambiente Virtual de Ensino - Aprendizagem. www.cead.ifes.edu.br	10
6. Utilizando vetores e matrizes	Diferenciar vetores e Matrizes. Demonstrar a praticidade de utilização de vetores. Apresentar as vantagens de usar *arrays*.	Caderno e Ambiente Virtual de Ensino -Aprendizagem. www.cead.ifes.edu.br	10
7. Manipulando arquivos	Demonstrar a importância do armazenamento e recuperação de dados. Enumerar os aspectos fundamentais para a leitura e gravação em arquivos. Apresentar os passos necessários para armazenar arquivos no formato texto.	Caderno e Ambiente Virtual de Ensino-Aprendizagem. www.cead.ifes.edu.br	10
8. Estrutura de dados em Java – listas	Demonstrar a importância do armazenamento e recuperação de estruturas de dados. Enumerar os aspectos fundamentais para a utilização de listas lineares. Apresentar os passos necessários para implementação de listas.	Caderno e Ambiente Virtual de Aprendizagem. www.cead.ifes.edu.br	10
9. Estrutura de dados em Java – pilha	Demonstrar a importância do armazenamento e recuperação utilizando estruturas de dados. Enumerar os aspectos fundamentais para a utilização de pilhas. Apresentar os passos necessários para implementação de pilhas	Caderno e Ambiente Virtual de Aprendizagem. www.cead.ifes.edu.br	10

Aula 1 – Plataforma Java

Objetivos

Descrever as principais características da linguagem Java.

Descrever os procedimentos necessários para o desenvolvimento de uma aplicação Java.

Fornecer o primeiro contato com a linguagem Java.

1.1 Introdução

A linguagem Java surgiu por acaso quando quem a criou, uma equipe de engenheiros da empresa Sun, foi tomar um café na esquina. Como numa reunião de amigos, esses engenheiros assim a denominaram, inspirados pelo nome da cidade de onde provinha o café que tomavam.

Na realidade, não existe um consenso entre os escritores a respeito da verdadeira história dessa linguagem. Alguns autores afirmam que o nome Java se deve a um passeio que o pessoal da Sun fez numa ilha da Indonésia com esse mesmo nome.

Originalmente, a linguagem foi criada para ser utilizada em pequenos equipamentos eletrônicos; entretanto, com pouco recurso financeiro desse setor na época e principalmente com o aparecimento da internet, novas oportunidades apareceram e a empresa Sun passou a se dedicar a essa área.

Em 1995, a Sun anunciou Java não apenas como mais uma linguagem de programação, mas como uma nova plataforma de desenvolvimento. Dessa forma, a linguagem Java começou a ser utilizada para elaborar páginas da internet, proporcionando conteúdos interativos e dinâmicos, iniciando com a utilização de *applets* com imagens em movimento.

1.2 A linguagem Java

Essa linguagem tem tido muito sucesso no mercado e diversas ferramentas têm surgido para manipular ou gerar código Java. A própria Microsoft manteve o **Visual J++** como uma de suas ferramentas de desenvolvimento, aparentemente sem muito sucesso, sofrendo vários processos por parte da Sun, o que provocou seu desaparecimento na nova versão do Microsoft Studio. Praticamente todos os principais fabricantes de *software* sentiram a necessidade de lançar no mercado alguma ferramenta para manipular Java,

Visual J++
Foi a implementação específica da Microsoft para o Java, em inglês pronuncia-se "J*ay plus plus*". Otimizado para a Plataforma Windows, os programas de J++ poderiam funcionar somente no MSJVM (Máquina Virtual Java da Microsoft), que foi a tentativa da Microsoft para criar um interpretador mais rápido. A sintaxe, *keywords*, e convenções gramaticais eram os mesmos do Java.

o que mostra sua força e longevidade para os próximos anos no ambiente das linguagens de programação mais usadas.

Hoje, quando Java é mencionado, deve-se entendê-la de imediato como a linguagem da Sun, a empresa que a fez nascer e a mantém como uma marca registrada. A linguagem Java da Sun tem feito muito sucesso, e uma das coisas que a torna tão atraente é o fato de que programas escritos em Java podem ser executados virtualmente em qualquer plataforma, aceitos em qualquer tipo de computador (ou outros aparelhos), características marcantes da internet. Com Java o processamento pode deixar de ser realizado apenas no lado do servidor, como era a internet no princípio, passando a ser executado também no cliente (entenda-se *browser*).

O aspecto da utilização de Java em multiplataforma é muito importante, porque os programadores não necessitam ficar preocupados em saber em qual máquina o programa será executado, uma vez que um mesmo programa pode ser usado num PC, num Mac ou em um computador de grande porte. É muito melhor para uma empresa desenvolver um *software* que possa ser executado em "qualquer lugar", independentemente da máquina do cliente.

Java pode atuar em conjunto com outras linguagens, como é o caso de HTML, em que as aplicações podem ser embutidas em documentos HTML, podendo ser transmitidas e utilizadas na internet. Os programas escritos em Java funcionam como um acessório (chamado de ***applet***) que é colocado no computador do usuário no momento que ele acessa um *site* qualquer, isto é, o computador do usuário passa a executar um programa armazenado no servidor *web* que é transferido para sua máquina no momento do acesso.

A-Z

Applet

É um software aplicativo que é executado no contexto de outro programa (como, por exemplo, um web browser), um applet geralmente executa funções bem específicas. O termo foi introduzido pelo AppleScript em 1993 – dados e gerador de relatórios. No contexto de Java, applets são aplicativos que se servem da Java Virtual Machine (JVM) existente na máquina do cliente ou embutida no próprio navegador do cliente para interpretar o seu bytecode

Num certo *site* o usuário pode executar um programa para a compra de um veículo e, logo em seguida, ao acessar outro *site*, executar outro programa para consultar o extrato bancário; tudo escrito em Java e executado em sua máquina local.

A linguagem Java também tem sido usada para a criação dos processos automáticos na *web*. Os processos envolvidos na atualização de notícias, por exemplo, aqueles que aparecem a cada minuto em um *site* qualquer, são aplicações desenvolvidas a partir do Java.

Outro aspecto a ser observado sobre a linguagem Java é sua semelhança com a **linguagem C++**, tanto no que diz respeito à sintaxe dos comandos utilizados quanto na característica de ser orientada a objetos. A programação orientada a objetos é hoje universalmente adotada como padrão de mercado, e muitas linguagens tradicionais foram aperfeiçoadas para implementar essa nova forma de trabalho; Java já nasceu assim.

A-Z

Linguagem C++
Pode-se dizer que C++ foi a única linguagem, entre tantas outras, que obteve sucesso como uma sucessora à linguagem C, inclusive servindo de inspiração para outras linguagens como Java e IDL de CORBA.

O grande diferencial de Java em relação às outras linguagens de programação se refere ao fato de que ela foi concebida, originalmente, para ser usada no ambiente da *World Wide Web* (WWW). Nos últimos cinco anos, a grande maioria das linguagens tem buscado se adaptar a essa nova realidade e necessidade; entretanto, Java é a que mais tem se destacado até o momento.

1.3 As características da linguagem Java

A linguagem Java possui diversas características, entre as quais podemos destacar:

a) **Orientação a objetos**: é uma prática de programação já sólida no mercado, e a maioria das linguagens de hoje permite trabalhar dessa forma. Como conceito inicial, imagine a orientação a objetos como uma prática de programação que permite a utilização de diversos trechos de código. Esses objetos podem simular um objeto do mundo real, como um automóvel, uma casa, uma pessoa etc.

b) **Portabilidade**: Java é uma linguagem multiplataforma, ou seja, uma mesma aplicação pode ser executada a diferentes tipos de plataforma sem a necessidade de adaptação de código. Essa portabilidade permite que um programa escrito na linguagem Java seja executado em qualquer sistema operacional.

c) ***Multithreading***: *threads* (linhas de execução) é o meio pelo qual se consegue fazer com que mais de um evento aconteça, simultaneamente, em um programa. Assim, é possível criar servidores de rede multiusuários, em que cada *thread*, por exemplo, cuida de uma conexão de um usuário ao servidor, isto é, um mesmo programa.

d) **Suporte à comunicação**: uma das vantagens de Java é fornecer um grande conjunto de classes com funcionalidades específicas, ou seja, muitos detalhes de programação são encapsulados em classes já prontas. Nesse contexto, a linguagem oferece um conjunto de classes para programação em rede, o que agiliza a implementação.

e) **Acesso remoto a banco de dados**: possibilita que dados sejam recuperados e/ou armazenados de qualquer ponto de internet. Essa é uma característica muito importante, se considerado o grau de automação proporcionado pelo Java.

Um aspecto importante que deve ser levado em consideração, principalmente porque o próprio mercado afirma, refere-se aos mecanismos de segurança que a linguagem oferece para a realização de processos pela internet. Se comparada a outras linguagens usadas na internet, como ASP, por exemplo, Java possui maior segurança, com diversas classes que tratam de chaves públicas e privadas para a geração de dados criptografados.

1.4 Criação de programas em Java

Para a criação de programas em Java, torna-se necessária a digitação por meio de uma ferramenta específica ou ainda de um editor de textos qualquer, gerando o código-fonte do programa.

Depois de digitado, esse programa deve passar por um processo de análise do código, a fim de que seja verificada a existência de erros de sintaxe. Esse processo é chamado de compilação e é realizado por meio de um compilador Java, normalmente o compilador do *kit* de desenvolvimento da Sun. Todo programa Java deve ser compilado, assim como ocorre com linguagens de programação como Pascal, C, entre outras.

Com o compilador é realizada a tradução do programa escrito em Java para uma linguagem intermediária chamada Java *bytecodes*, um código independente de plataforma que é decifrado por um interpretador Java; isto é, para que um programa em Java seja executado, é necessário possuir outra ferramenta chamada interpretador. O interpretador é o responsável por executar o programa escrito em Java em que cada instrução do *bytecode* é interpretada, sendo executada no computador.

A Figura 1.1 ilustra a sequência de desenvolvimento de um programa em Java, como este deve ser criado na forma de uma classe. Conforme pode ser observado, uma classe em Java (código-fonte) pode ser digitada em um editor de textos qualquer e deve ser salva com a extensão Java.

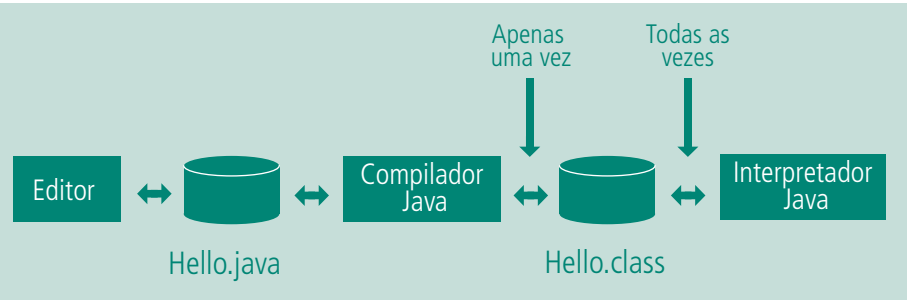


Figura 1.1: Sequência de desenvolvimento de um programa em Java
Fonte: http://java.sun.com/javase/downloads/index.jsp

A seguir uma ferramenta realiza sua compilação (compilador). Caso ocorram erros no processo de compilação, o programa-fonte deve ser corrigido e compilado novamente enquanto persistirem os erros. Quando não existirem mais erros de complicação, será gerado um arquivo com extensão ".class" (o arquivo com os ***bytecodes***), a ser executado por um interpretador Java ou pelo *browser*, caso o programa seja utilizado na internet. Na maioria das principais ferramentas de desenvolvimento, o processo de compilação é automático, isto é, ocorre durante a digitação do código-fonte, ou seja, a compilação vai sendo executada automaticamente durante a digitação da mesma forma que o corretor ortográfico dos editores de texto atuais.

A-Z

Bytecodes
São gerados pelo processo de compilação, específicos a qualquer máquina física, são instruções para uma máquina virtual.

1.5 A plataforma Java

Plataforma é um ambiente de *software* ou *hardware* no qual um programa roda. A maioria das plataformas é formada pelo conjunto *hardware* e um sistema operacional, isto é, um conjunto de *hardware* e *software* que atuam juntos. Java difere da maioria das outras plataformas porque é composta apenas de um *software* operando com outra plataforma qualquer.

No mundo dos computadores existem muitas plataformas, como Microsoft Windows, Macintosh, OS/2, Unix e netware. Normalmente, para que um mesmo programa funcione em diferentes plataformas, é necessário que ele seja compilado separadamente; isto é, ele deve ser compilado na plataforma em que será executado. Uma aplicação que é executada sobre uma plataforma pode não funcionar sobre outra, porque o arquivo foi criado para uma plataforma específica.

Servlet é um componente do lado servidor que gera dados HTML e XML para a camada de apresentação de um aplicativo Web. É basicamente uma classe na linguagem de programação Java que dinamicamente processa requisições e respostas, proporcionando dessa maneira novos recursos aos servidores. A definição mais usada considera-o extensão de servidores.

Java é uma nova plataforma de *software* que possibilita que um mesmo programa seja executado em diversas plataformas, talvez a característica mais importante dessa linguagem.

Um programa escrito na linguagem Java é compilado e gera um arquivo de *bytecodes* (com extensão .class), que pode ser executado onde quer que a plataforma Java esteja presente, em qualquer sistema operacional subjacente. Em outras palavras, o mesmo programa pode ser executado em qualquer sistema operacional que execute a plataforma Java. Uma analogia relacionada à plataforma Java pode ser visualizada na Figura 1.2.

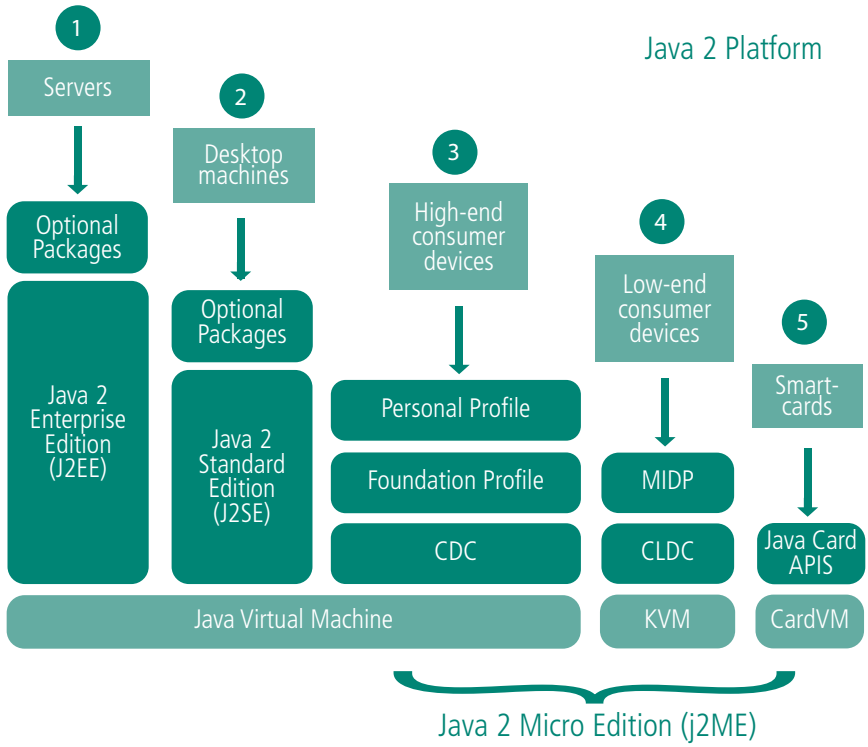


Figura 1.2: Plataforma Java
Fonte: Furgeri (2008)

Enquanto cada plataforma possui sua própria implementação da máquina virtual Java, existe somente uma especificação padronizada para a máquina virtual, proporcionando uma interface uniforme para aplicações de qualquer *hardware*. Máquina Virtual Java (*Java Virtual Machine*) é ideal para uso na internet, em que um programa deve ser executado em diferentes máquinas pela *web*.

1.6 Ambiente de desenvolvimento

Nesta seção apresentamos um esboço das ferramentas necessárias para a elaboração, compilação e execução de aplicações em Java. Como qualquer outra imagem, há muitas opções no mercado e diversas maneiras de desenvolver em Java. Existe uma infinidade de ferramentas que pode deixar o desenvolvedor em dúvida na hora de escolher o ambiente de trabalho. No momento, as ferramentas que mais se destacam são Netbeans e Eclipse; no entanto, existem muitas outras.

A ferramenta JDK da Sun é composta basicamente por um compilador (javac), um interpretador (Java), um visualizador de *applets* (*appletviewer*), bibliotecas de desenvolvimentos (*packages*), um depurador de programas (JDB) e diversas documentações (javadoc). Essa ferramenta da Sun não fornece um ambiente visual de desenvolvimento, porém trata-se do principal padrão a ser seguido, visto a enorme funcionalidade que possui aliada á facilidade de utilização.

Para a digitação de código-fonte das classes em Java, a única ferramenta necessária é o bloco de notas do Windows; entretanto, qualquer editor de textos disponível na máquina do leitor pode ser utilizado.

1.6.1 Instalação de *kit* de desenvolvimento da Sun

Uma das maiores dificuldades dos iniciantes em Java é conseguir instalar corretamente o *kit* de ferramentas da Sun, uma vez que nem todo o processo ocorre de forma automática como na maioria dos instaladores de *software*. Por esse motivo, é importante dedicar um tempo a esse processo, mesmo sabendo que existem muitas variações, dependendo do sistema operacional em que a ferramenta será instalada.

Antes de desenvolver as aplicações em Java, é necessário possuir instaladas em sua máquina todas as ferramentas de desenvolvimento. Por isso, apresentamos a instalação das ferramentas mínimas necessárias à criação de aplicações em Java.

A Sun fornece *download* gratuito de sua ferramenta no endereço **http://java.sun.com/javase/downloads/index.jsp**, em que são encontradas versões para várias plataformas. O nome do *kit* de ferramentas que você deve baixar é "JDK 6 *update* 21", ou ainda uma outra versão mais recente, caso se encontre disponível (Figura 1.3).

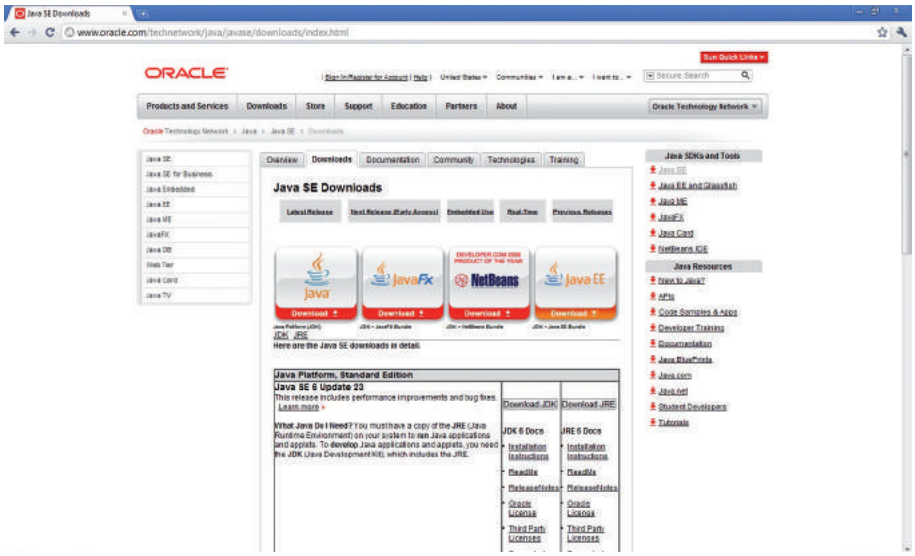


Figura 1.3: *Site* para baixar instalação do pacote da Sun
Fonte: http//java.sun.com/javase/downloads/index.jsp

Os procedimentos para a correta instalação da ferramenta variam de acordo com a plataforma em que será instalada e também em função da versão do sistema operacional.

1.6.2 Instalação do JDK na plataforma Windows

Os procedimentos para instalação do JDK no Windows são os seguintes:

1. Faça o *download* da versão correspondente ao Windows.

O processo de instalação transfere todas as ferramentas e pacotes da linguagem para sua máquina. Ao instalar o JDK, é criada uma pasta com todos os arquivos do *kit* de ferramentas da Sun. O caminho *default* da instalação é C:/arquivos de programas\java\jdk1.6.0_21". Dependendo da versão instalada, uma pasta de nome diferente será criada.

Provavelmente, a Sun disponibilizará outras versões em breve como, por exemplo, jdk 1.6.0_04, jdk 1.6.0_05 e assim por diante. Os números 04 ou 05 ao final do nome normalmente se referem ao *update*. Ao instalar o JDK, são criadas diversas pastas, como as mostradas na Figura 1.4.

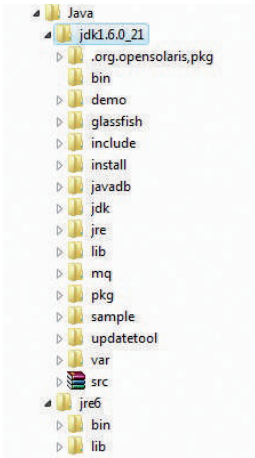


Figura 1.4: Localização da pasta de instalação
Fonte: ***Printscreen*** Windows 2000 e XP

Observe que a pasta jdk 1.6.0_21 é a principal em que estão todas as outras (a Sun chama-a de JAVA_HOME). Dependendo da versão do JDK instalada, essa pasta pode ter nomes e conteúdos diferentes. Você deverá se concentrar no nome da versão que baixou.

2. Realize as configurações das variáveis de ambiente, as quais dependem do sistema operacional em que você está instalando o JDK. Os procedimentos apresentados em seguida se referem à configuração para o ambiente Windows.

No Windows 2000 e XP, devemos configurar as variáveis pelo painel de controle. Defina as variáveis seguindo os procedimentos:

a) Acesse o painel de controle.

b) Abra o item sistema.

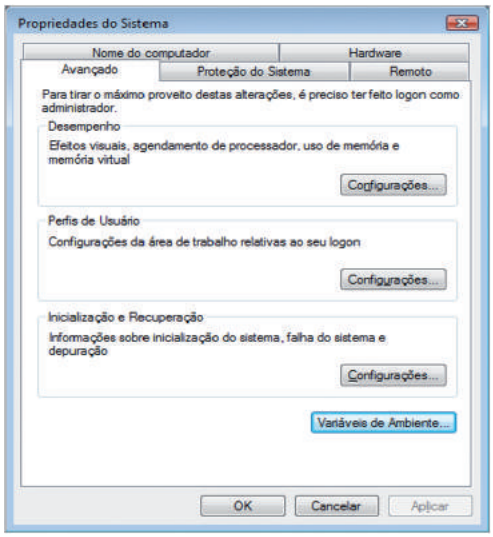


Figura 1.5: Janela de configuração da variável de ambiente
Fonte: *Printscreen* Windows 2000 e XP

3. Clique na guia "avançado" e em seguida, no botão "variáveis de ambiente".

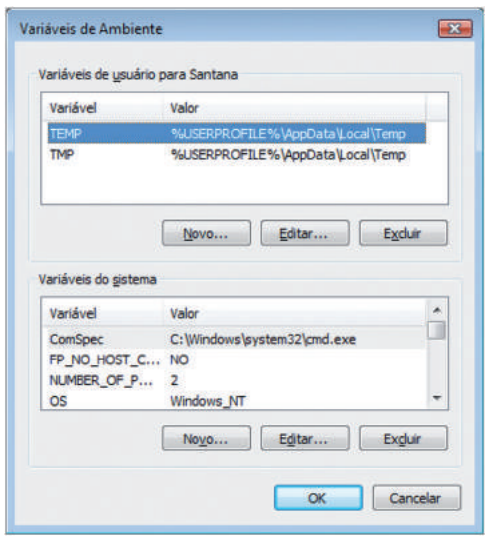


Figura 1.6: Janela de configuração da variável de ambiente
Fonte: *Printscreen* Windows 2000 e XP

4. Em "variáveis de usuário", clique no botão "nova". As variáveis de ambiente podem também ser definidas em "variáveis do sistema" em vez de "variáveis de usuário", como sugerido. A diferença é que, quando definidas em "variáveis de usuários", elas funcionam somente para o seu usuário, e em "variáveis de sistema" funcionam para todos os usuários.

5. Surge a janela "nova variável de usuário". No campo "nome da variável" coloque o nome da variável que será incluída, por exemplo, **JAVA_HOME,** e no campo "valor da variável" coloque o caminho referente à variável que você nomeou, "C:\arquivo de programas\java\jdk1.6.0_21" (sem as aspas), e clique no botão "OK".

6. Faça o mesmo procedimento de inclusão com as variáveis **path** e **classpath**, definindo os seus nomes e incluindo os valores correspondentes (Figura 1.7):

"C:\arquivos de programas\java\jdk1.6.0_21\bin" para a variável **path** e "C:\arquivos de programas\java\jdk1.6.0_21\lib;.;" para a variável **classpath**.

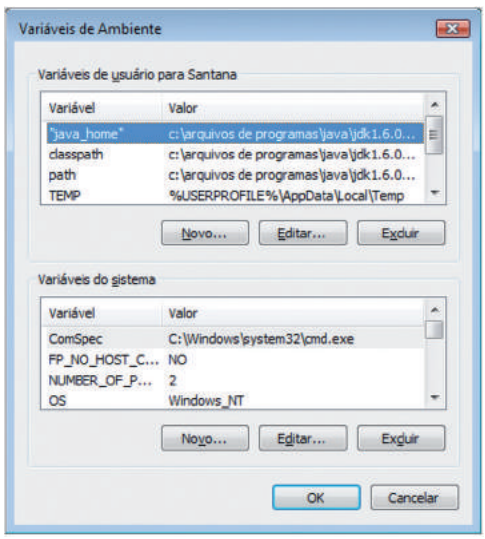


Figura 1.7: Janela de configuração da variável de ambiente
Fonte: *Printscreen* Windows 2000 e XP

7. Verifique se a instalação foi realizada com sucesso. Entre em um diretório qualquer no *prompt* de comando e digite "javac" seguido da tecla *Enter*. Se aparecer uma mensagem como "javac" não é reconhecido como um comando interno", é porque o Java não foi instalado corretamente Se isso ocorrer, refaça a configuração, verificando principalmente as configurações das variáveis de ambiente. Se ao digitar "javac" aparecer uma tela com instruções de *help* do Java, significa que a instalação e a configuração foram realizadas com sucesso.

Como sugestão de *link* para instalação do Java, utilize o endereço: http://www.youtube.com/watch?v=wvzUm0ys0vM&feature=related

Faça a instalação da última atualização do Java em seu computador e verifique se funciona.

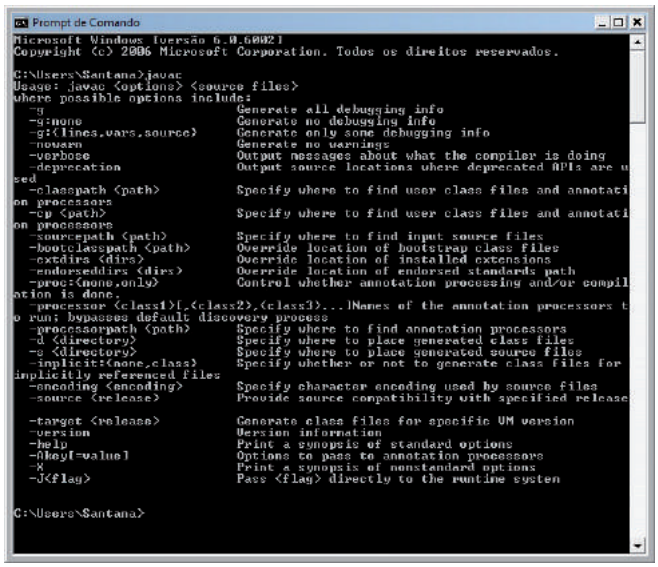


Figura 1.8: Janela de verificação da instalação do Java
Fonte: *Printscreen* Windows 2000 e XP

Para o Windows Vista/7:

a) Acesse o painel de controle.

b) Abra o item "sistema e manutenção". Para Windows 7 abra "sistema e segurança".

c) Abra o item "sistema".

d) Ao lado esquerdo, no *menu* "tarefas", clique em "configurações avançadas do sistema", em seguida no botão "variáveis de ambiente". Continue executando o passo 4 anterior referente ao Windows XP/2000.

Observações:

- Em função de constantes atualizações do JDK, o leitor deve atentar para o número da versão da ferramenta carregada no processo de *download*, de maneira a fazer sua correta instalação. O mesmo nome da pasta principal (exatamente o mesmo nome) deve ser inserido nas variáveis de ambiente. O trecho em negrito no código anterior será variável, dependendo da versão instalada.

- Atenção especial deve ser dada quando a instalação for realizada no Windows Vista. Se o *firewall* nativo estiver ativado, talvez seja necessário desbloquear o Java. Se o Java estiver bloqueado, pode ocorrer um erro referente à violação de acesso por parte da máquina virtual quando ele for executado.

- Outro ponto ser observado com relação ao Vista refere-se à configuração das variáveis de ambiente, as quais tiveram de ser definidas com o caminho reduzido do modo MS-DOS> por exemplo, o conteúdo da variável **path** ('c:\arquivos de programas\java\jdk1.6.0_03\bin") foi definido como "C:\arquiv~1\java\jdk1.6.0\bin": caso contrário, o compilador não seria localizado. O mesmo procedimento foi realizado para as variáveis **JAVA_HOME** e **classpath**.

1.7 Primeiro contato com o Java

Inicialmente, para fornecer o primeiro contato com a linguagem, será apresentada uma classe em Java que escreve uma mensagem qualquer na tela. Apesar de o exemplo ser simples, ele contém os itens fundamentais para a criação de qualquer aplicação em Java: elaboração do código, compilação e execução. Esses itens serão seguidos durante o processo de elaboração das aplicações.

Como sugestão, utilize o bloco de notas, um editor simples e rápido que atende a todos os requisitos míninos para a construção de aplicações em Java.

Todo programa em Java inicia com a palavra reservada *class* seguida do nome da classe (no caso Exemplo01). Como convenção definida pela Sun, todo nome de classe inicia com letra maiúscula.

Digite o nome do programa (class Exemplo01) mostrado na Figura 1.9 e salve-o com esse mesmo nome em uma pasta.

Como sugestão de *link*, utilize o endereço: http://www.youtube.com/watch?v=xGM9uBOvMgQ&feature=related

Exemplo01.java

```
1  class Exemplo01
2  {
3      Public static void main (String args[])
4      {
5          System.out.println("programa exemplo1");
6      }
7  }
```

Figura 1.9: Exemplo01
Fonte: Elaborada pelo autor

Crie um programa em Java para mostrar seu nome impresso na tela.

Um par de chaves envolve todo o código da classe sempre; um programa em Java possui uma classe que envolve o código.

Uma classe em Java é composta por métodos (considerados funções ou *procedures* em outras linguagens de programação) que podem conter outras estruturas de programa. Toda classe executável, ou seja, toda classe que será interpretada e executada, deve obrigatoriamente possuir o método *main* (principal), que é invocado quando a classe é executada.

Nesse caso, quando a classe for executada, será invocado o método *main* que possui duas instruções para envio de mensagens na tela (system.out.println). Não é exatamente uma instrução e sim uma classe da linguagem especializada em saída de dados.

Observações:

- A linha "public static void main" (*string args*) aparece em todas as classes executáveis nesse mesmo formato.

- A linha do método principal possui o seguinte formato: "public static void main" (*string args*[]); praticamente todas as aplicações têm essa linha e é a variável **args** que pode receber outro nome de acordo com o desejo do programador.

Resumo

Nesta aula falamos sobre os conceitos iniciais de Java, como configuração e instalação. Abordamos o conceito de programação em Java (como variáveis de ambiente, classes, padrões, etc.), assim como o funcionamento do seu ambiente de desenvolvimento.

Atividades de aprendizagem

1. Por que a utilização de Java em multiplataforma é muito importante para programadores?

2. Qual a principal característica que distingue a plataforma Java das demais existentes?

Aula 2 – Aspectos fundamentais sobre Java

Objetivos

Realizar o primeiro contato com Java.

Demonstrar declaração de dados.

Verificar os conversores de tipo em Java.

2.1 Tipos de dados

Assim como em outras linguagens, antes de utilizar variáveis é necessário saber definir um tipo de dado. Os tipos de dados em Java são portáteis entre todas as plataformas de computadores que suportam essa linguagem.

Na maioria das linguagens, quando um dado inteiro é utilizado, pode ser que para uma determinada plataforma esse número seja armazenado com 16 *bits* e em outra 32 *bits*. Em Java isso não ocorre, uma vez que um tipo de dado terá sempre a mesma dimensão.

Os tipos primitivos da linguagem (Quadro 2.1) utilizados na criação de variáveis são:

a) *Boolean*: não é um valor numérico, só admite os valores *true* ou *false*.

b) *Char*: usa o código UNICODE e ocupa cada caractere 16 *bits*.

c) Inteiros: diferem nas precisões e podem ser positivos ou negativos.

- *Byte*: 1 *byte*.
- *Short*: 2 *bytes*.
- *Int*: 4 *bytes*.
- *Long*: 8 *bytes*.

d) Reais em ponto flutuante: igual aos inteiros, também diferem nas precisões e podem ser positivos ou negativos.

- *Float*: 4 *bytes*.
- *Double*: 8 bytes.

Quadro 2.1: Os tipos primitivos em Java

Tipo de dados	Definição	Tipo	Tamanho (bits)	Exemplos
Literal (caractere)	Letras, números e símbolos	**char**	16	'a', '?', '*'
Inteiro	Números inteiros positivos ou negativos	**byte**	8	0, 1, 23
		int	32	0, 1, 23
		short	16	0, 1, 23
		long	64	0, 1, 23
Real (ponto flutuante)	Números com casas decimais, positivos ou negativos	**float**	32	0.34 8.65
		double	64	0.34 8.65
Lógico (Booleano)	Verdadeiro (1) ou falso (0)	**boolean**	8	true false

Fonte: Furgeri (2008)

A-Z

Identificador
É a localização da memória capaz de armazenar o valor de um certo tipo, para o qual se dá um nome que descreve seu significado ou propósito.

2.2 Definição de variáveis e constantes

Uma variável ou constante é um tipo de **identificador** cujo nome, que é selecionado pelo programador, é associado a um valor que pertence a um tipo de dado.

Todo identificador possui um nome, um tipo e conteúdo. Os identificadores não podem utilizar palavras reservadas do Java.

A linguagem Java exige que os identificadores tenham um tipo de dado definido antes de serem utilizados no programa, ou seja, eles devem ser obrigatoriamente declarados, independentemente do ponto do programa, seja no meio, no início ou no final, desde que antes de sua utilização no programa.

Essa característica do identificador em Java difere da maioria das linguagens de programação. A linguagem Pascal, por exemplo, possui um local exclusivo para declaração de variáveis.

Uma variável precisa ser declarada para poder ser utilizada. Opcionalmente, ela pode ser inicializada já no momento de sua declaração. O código da Figura 2.1 mostra alguns exemplos de manipulação de variáveis em Java.

Exemplo02.java

```
1  public class Exemplo02 {
2      public static void main (String args[ ]) {
3      int n1; /* Declaração de um inteiro. */
4      int n2 = 4; /* Declaração e inicialização de outro inteiro. */
5      char c = 'x'; /* Declaração e inicialização de um caractere. */
6      n1 = n2 + 8; // Atribuindo valor à n1
7      System.out.println("Primeiro valor: " + n1);
8      System.out.println("Segundo valor: " + n2);
9      System.out.println("Terceiro valor: " + c);
10     }
11 }
```

Figura 2.1: Exemplo02
Fonte: Elaborada pelo autor

Como você pôde perceber no último exemplo, um comentário em Java pode ser escrito com // (para comentar apenas até o final da linha) ou com /* */ (para comentar tudo o que estiver entre o /* e o */).

Para identificar a documentação, utilizamos /** */ A saída do programa Exemplo02 deverá ser:

12
4
x

Caso uma variável do tipo *char*, *byte*, *short*, *int*, *long*, *float* ou *double* não seja inicializada, ela é criada com o valor 0. Se ela for do tipo *boolean*, seu valor padrão será *false*.

Quando for necessário definir uma nova variável com um tipo de dado diferente, por convenção, utiliza-se uma nova linha. O mais comum entre os programadores Java é definir um tipo de dados e declarar uma lista com um ou mais nomes de variáveis desejadas desse tipo. Nessa lista os nomes são separados por vírgulas e a declaração terminada por ' ; ' (ponto e vírgula).

É possível criar mais de uma variável do mesmo tipo na mesma linha, separando-as por uma vírgula. Exemplo: int x, y, z;

As variáveis também podem ter sensibilidade, isto é, ao declarar uma variável com um nome (por exemplo, **dolar**) ele deve ser utilizado sempre da mesma forma. Isto é, não pode ser usado como Dólar, DOLAR, dólar ou qualquer outra variação, apenas com todas as letras minúsculas, como realizado em sua declaração.

Os nomes das variáveis devem começar com letra, caractere de sublinhado ou cifrão. Não é permitido iniciar o nome da variável com número. Por convenção, a linguagem Java utiliza o seguinte padrão:

- quando o nome da variável for composto apenas por um caractere ou palavra, os caracteres devem ser minúsculos;

- quando o nome da variável tiver mais de uma palavra, a primeira letra da segunda palavra em diante deve ser maiúscula. Todos os outros caracteres devem ser minúsculos.

Exemplos: a, a1, real, nome, valorVenda, codigoFornecedor.

Outro ponto a ser observado se refere à utilização do ponto e vírgula (;) no final da maioria das linhas de código.

2.3 Declaração de constantes

Na realidade não existem constantes em Java; o que existe é um tipo de variável com comportamento semelhante a uma constante de outras linguagens. Trata-se de um tipo de variável que não pode alterar seu conteúdo depois de ter sido inicializado, ou seja, o conteúdo permanece o mesmo durante toda execução do programa. Em Java, essa variável é chamada **final**. Essas constantes são usadas para armazenar valores fixos, geralmente, definidos no início de uma classe. Por convenção os nomes de constantes devem ser escritos em letras maiúsculas. Exemplos: na Matemática temos a constante **PI** cujo valor é 3,1416 (isto é, p=3,1416); na Física temos o valor da aceleração da **GRAVIDADE** da Terra (g=9,81 m/s^2).

Para a declaração de constantes em Java utiliza-se a palavra reservada **final** antes da definição do tipo de variável:

```
final double PI=3.14;
final double GRAVIDADE=9.81;
```

Caso um segundo valor seja atribuído a uma variável final no decorrer da classe, o compilador gera uma mensagem de erro. Não é obrigatório inicializar o conteúdo de uma variável final no momento de sua declaração

2.4 Comentários

Os comentários são linhas adicionadas ao programa que servem para facilitar seu entendimento por parte do programador, ou ainda por outra pessoa que o consulte. Essas linhas não afetam o programa em si, pois não são consideradas parte do código. O Java aceita três tipos de comentário: de linha, de múltiplas linhas e de documentação.

Para inserir comentários de linha única, utiliza-se // (duas barras) em qualquer local do programa e tudo o que tiver escrito depois desse sinal e na mesma linha será considerado um comentário.

Para inserir comentários que envolvam várias linhas, utiliza-se /* (barra asterisco) para marcar o início e */ (asterisco barra) para o final, ou seja, tudo o que estiver entre esses dois sinais será considerado comentário.

O terceiro tipo é semelhante ao comentário de múltiplas linhas; entretanto, tem o propósito de possibilitar a documentação do programa por meio de um utilitário (javadoc) fornecido pela Sun junto com o SDK.

Verifique o Exemplo21 (Figura 2.2) e sua execução (Figura 2.3). Os comentários não aparecem na execução do programa.

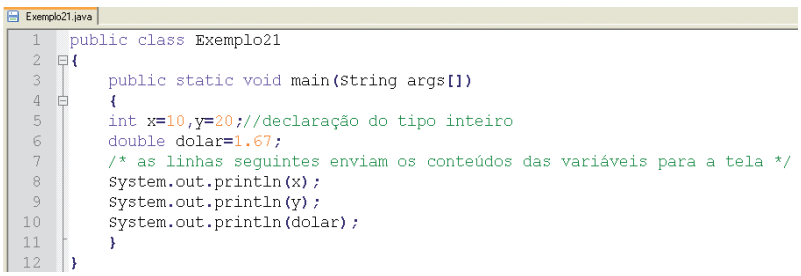


Figura 2.2: Exemplo21
Fonte: Elaborada pelo autor

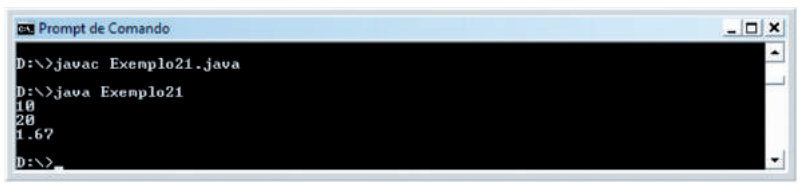


Figura 2.3: Execução do programa Exemplo21
Fonte: Elaborada pelo autor

2.5 Operadores

A linguagem Java oferece um amplo conjunto de operadores destinados à realização de operações aritméticas, lógicas e relacionais, com a possibilidade de formar expressões de qualquer tipo. Além dos operadores matemáticos, existem também operadores lógicos e relacionais.

2.5.1 Operadores aritméticos

Entre os operadores presentes no Quadro 2.2, talvez os decremento (--) e o incremento (++) causem alguma dúvida, principalmente para os programa-

dores iniciantes. Entretanto sua utilização é extremamente simples: o operador de incremento aumenta o valor de uma variável qualquer em um. O mesmo vale para o operador de decremento, logicamente, reduzindo em um o valor da variável.

Quadro 2.2: Operadores aritméticos

Operação	Sinal	Exemplo
Adição	+	1+20
Subtração	-	35-17
Multiplicação	*	14*2
Divisão	/	14/2
Resto da divisão inteira	%	14%7
Sinal negativo	-	-4
Sinal postitivo	+	+5
Incremento unitário	++	++6 ou 6++
Decremento unitário	--	--6 ou 6--

O Exemplo0203 mostra um programa em Java com a utilização de alguns operadores (Figuras 2.4 e 2.5).

Exemplo0203.java

```
1  class Exemplo0203
2  {
3    public static void main (String args[])
4    {
5      // declaração e inicialização de variáveis
6      int x = 10;      int y = 3;
7      // várias operações com as variáveis
8      System.out.println("X = "+ x);
9      System.out.println("Y = "+ y);
10     System.out.println("-X = "+(-x));
11     System.out.println("X/Y = "+(x/y));
12     System.out.println("Resto de X por Y = "+ (x%y)); // resulta 1
13     System.out.println("Inteiro de X por Y = "+ (int)(x/y)); // resulta 3
14     System.out.println("X + 1 = "+ (++x)); // resulta 11
15   }
16 }
```

Figura 2.4: Exemplo0203

Fonte: Elaborada pelo autor

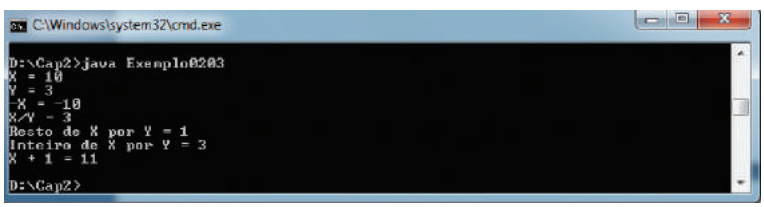


Figura 2.5: Execução do programa Exemplo0203

Fonte: Elaborada pelo autor

2.5.2 Operadores relacionais

Os operadores relacionais possibilitam comparar valores ou expressões, retornando um resultado lógico verdadeiro ou falso. O Quadro 2.3 mostra os operadores relacionais usados em Java e sua aplicação.

Quadro 2.3: Operadores relacionais em Java		
Significado	**Operador**	**Exemplo**
Igual	==	x==20
Diferente (Não igual)	!=	y!=17
Menor que	<	x<2
Maior que	>	x>2
Menor ou igual	<=	y<=7

2.5.3 Operadores lógicos

São operadores que permitem avaliar o resultado lógico de diferentes operações aritméticas em uma expressão. Os operadores lógicos usados em Java são mostrados no Quadro 2.4 a seguir.

Quadro 2.4: Operadores lógicos em Java		
Significado	**Operador**	**Exemplo**
Operação lógica E (AND)	&&	(x<5)&&(x>0)
Operação lógica OU (OR)	||	(y==5||y>10)
Negação	!	!true==false

2.6 Passagem de parâmetros

Foi apresentado no início do material que o método *main* recebe String args[] como parâmetro:

```
public static void main (String args[]) { … }
```

Como o *main* é o método principal, seu parâmetro é também parâmetro para o programa todo. String args[] é um vetor de *strings* formado por todos os argumentos passados ao programa na linha de comando do sistema operacional quando o programa é invocado. Para utilizá-los, basta acessar cada posição do vetor, como no Exemplo0204 mostrado na Figura 2.6.

```
Exemplo0204.java
1  class Exemplo0204
2  {
3    public static void main (String args[])
4    {
5      System.out.println(args[0]); // imprime o 1° argumento na tela
6      System.out.println(args[1]); // imprime o 2° argumento na tela
7    }
8  }
```

Figura 2.6: Exemplo0204
Fonte: Elaborada pelo autor

Para passarmos os parâmetros pela linha de comando, basta que adicionemos os valores após a linha de comando que utilizamos para executá-lo:

java nome-do-programa parametro1 parametro2 ...

Como exemplo, vamos considerar a linha de comando a seguir:

java Exemplo0204 Maria Fernanda

O resultado de sua execução está mostrado na Figura 2.7.

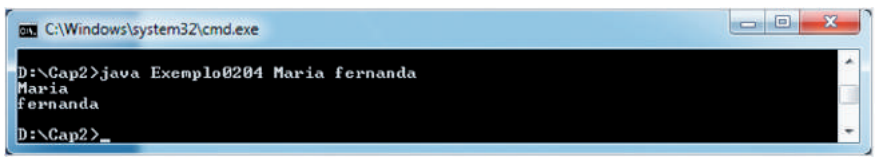


Figura 2.7: Execução do programa Exemplo0204
Fonte: Elaborada pelo autor

2.7 Conversão de tipos

É comum que o programador precise converter um número inteiro, por exemplo, em um número real (ou vice-versa). Em Java existem basicamente dois tipos de conversão de dados:

a) **conversão implícita** – na qual os dados são convertidos automaticamente, sem a preocupação do programador. Ela ocorre, por exemplo, quando convertemos um número inteiro para um número real.

Nesse caso, a conversão é implícita porque é óbvio para o compilador que um número inteiro pode ser representado também como um número real.

Veja um exemplo a seguir:

```
int x = 4;
float y = x;
double z = y;
```

b) conversão explícita – quando o programador precisa explicitar no código que um valor será convertido de um tipo para outro. No caso de um número real para um inteiro, por exemplo, pode haver perda na precisão do número.

Veja o exemplo a seguir:

```
float a = 9;
float b = a/8; // b = 1.125
int c = (int)b; /* Aqui estamos forçando a conversão para um número
inteiro. Nesse caso, a variável c armazenará apenas a parte inteira da
variável b, ou seja, 1 */
System.out.println(b);
System.out.println(c);
```

O resultado da execução deste trecho de código é:

```
1.125
1
```

O tipo *boolean* não pode ser convertido para nenhum outro tipo.

Seguindo o sentido das flechas da Figura 2.8 vemos os tipos que podem ser implicitamente convertidos em outros. Seguindo o sentido contrário, vemos os tipos que precisam ser convertidos explicitamente:

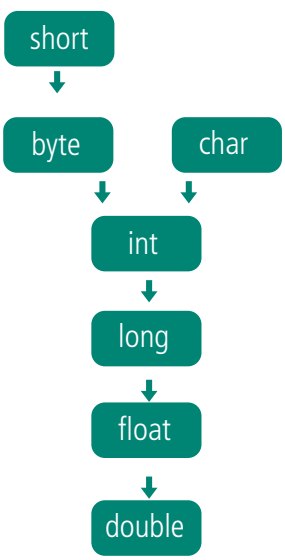


Figura 2.8: Conversões possíveis entre tipos primitivos em Java
Fonte: Elaborada pelo autor

Veja outro exemplo sobre utilização da conversão de tipos mostrado nas Figuras 2.9 e 2.10 a seguir.

Exemplo0205.java

```
1  class Exemplo0205
2  {
3    public static void main (String args[])
4    {
5      double nota1, nota2, trabalho, media;
6      nota1 = Double.parseDouble(args[0]);
7      nota2 = Double.parseDouble(args[1]);
8      trabalho = Double.parseDouble(args[2]);
9      media = (nota1 + nota2 + trabalho) / 3;
10     System.out.println("Media = "+ media);
11   }
12 }
```

Figura 2.9: Exemplo0205
Fonte: Elaborada pelo autor

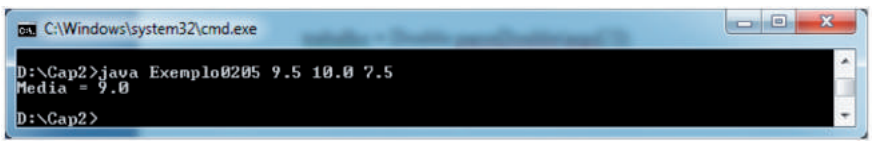


Figura 2.10: Execução do programa Exemplo0205
Fonte: Elaborada pelo autor

2.8 Entrada de dados pelo teclado

No Windows podemos fazer a entrada de dados pelo *prompt* de comando. No Linux, temos vários consoles, que são as telas em modo texto. Para enviarmos informações para o usuário pela saída padrão, utilizamos o método "**System.out.println**".

As aplicações de console utilizam como padrão a *stream* de dados *out*, disponível estaticamente na classe "**java.lang.System**". Uma *stream* pode ser entendida como um "duto" capaz de transportar dados de um lugar (um arquivo ou dispositivo) para outro. O conceito de *stream* é extremamente importante, pois é utilizado tanto para manipulação de dados existentes em arquivos quanto para comunicação em rede e outros dispositivos.

A *stream* de saída padrão é aberta automaticamente pela Máquina Virtual Java, ao iniciarmos uma aplicação Java, e permanece pronta para enviar dados. A saída padrão está tipicamente associada ao dispositivo de saída (*display*), ou seja, a janela de console utilizada pela aplicação conforme designado pelo sistema operacional.

Como fizemos em vários exemplos e exercícios anteriores, podemos enviar dados para a saída padrão utilizando o método "**System.out.println**". Da mesma forma que toda aplicação de console possui uma *stream* associada para ser utilizada como saída padrão, existe outra *stream* denominada entrada padrão, usualmente associada ao teclado do sistema. Essa *stream*, chamada *in*, está disponível estaticamente na classe "**java.lang.System**" e pertence à classe "**java.io.InputStream**", que também é aberta, automaticamente, quando a aplicação é iniciada pela máquina virtual Java (permanecendo pronta para fornecer os dados digitados).

Os métodos disponíveis para a entrada de dados na classe "**java.lang.System**" são bastante precários. Os três principais métodos são:

a) "read()": lê um *byte*;

b) "read(byte[])": preenche o *array* de *bytes* fornecido como argumento;

c) "skip(long)": descarta a quantidade de *bytes* especificada como argumento.

A leitura de *bytes* equivale à entrada de caracteres simples, o que é pouco confortável quando estamos trabalhando com valores numéricos (inteiros ou reais), *strings* ou outra informação diferente de caracteres, pois exige que cada caractere fornecido pelo usuário seja testado e concatenado com os demais para a formação de um determinado valor.

Para contornar essa situação, alguns materiais sugerem a criação de uma classe, que chamaremos aqui de Entrada, que contém três métodos para leitura de valores digitados pelo usuário:

a) "readDouble()": lê um valor *double* da entrada padrão;

b) "readInteger()": lê um valor inteiro da entrada padrão;

c) "readString()" lê uma *string* da entrada padrão.

No código da Figura 2.11, utilizamos a classe "**DatainputStream**" que pertence ao pacote "**java.io**". Entenda o pacote como um grupo de classes do mesmo tipo armazenadas em uma pasta qualquer. O asterisco presente em "import **java.io.***" indica que todas as classes do pacote "**java.io**" devem ser carregadas.

Exemplo0206.java

```
1  import java.io.*;
2  class Exemplo0206
3  {
4      public static void main (String args[])
5      {
6          String s = "";
7          float nota1 = 0, nota2 = 0, trabalho = 0, media = 0;
8          DataInputStream dado;
9          try
10         {
11             System.out.println("Entre com a nota 1");
12             dado = new DataInputStream(System.in);
13             s = dado.readLine();
14             nota1 = Float.parseFloat(s);
15
16             System.out.println("Entre com a nota 2");
17             dado = new DataInputStream(System.in);
18             s = dado.readLine();
19             nota2 = Float.parseFloat(s);
20             System.out.println("Entre com a nota do Trabalho");
21             dado = new DataInputStream(System.in);
22             s = dado.readLine();
23             trabalho = Float.parseFloat(s);
24             media = (nota1 + nota2 + trabalho) / 3;
25             System.out.println("Media : "+ media);
26         }
27         catch (IOException erro)
28         {
29             System.out.println("Houve erro na entrada de dados");
30         }
31         catch (NumberFormatException erro)
32         {
33             System.out.println("Houve erro na conversao, digite apenas caracteres
               numericos");
34         }
35     }
36 }
```

Figura 2.11: Classe DatainputStream
Fonte: Elaborada pelo autor

Resumo

Nesta aula falamos sobre os principais tipos de operadores em Java, como operadores lógicos e relacionais. Também abordamos o conceito de tipos de variáveis tão importante na programação em Java. Citamos também como funciona a entrada de dados em Java.

Atividades de aprendizagem

1. Crie um programa para apresentar mensagens, uma de boas-vindas, outra dizendo seu nome e mais uma informando sua idade.

2. Crie um programa que contenha duas variáveis de cada um dos tipos primitivos do Java. Coloque valores diferentes em cada uma delas e depois as imprima.

3. Altere três vezes o valor de uma das variáveis do programa anterior. Imprima a variável a cada nova atribuição.

4. Crie três variáveis do tipo *int* que contenham os valores 12, 13 e 14. Converta cada uma para um *float* e imprima o valor convertido. Qual o resultado impresso na tela?

5. Agora, crie três variáveis do tipo *float* que contenham os valores 12.3, 12.5 e 12.8. Converta cada uma para *int* e imprima o valor convertido. Qual o resultado impresso na tela?

6. Tente imprimir a soma de uma variável inteira com uma do tipo *float*. O que acontece?

7. Crie variáveis que contenham o primeiro termo e a razão de uma Progressão Aritmética, além de um inteiro *n* qualquer (que indique o número de termos dessa P.A.). Utilizando os valores criados, calcule o n-ésimo termo da progressão e a soma de seus **n** primeiros elementos.

8. Agora, crie variáveis que contenham o primeiro termo e a razão de uma Progressão Geométrica, e um inteiro **n** qualquer (que indique o número de termos dessa P.G.). Calcule o n-ésimo termo da progressão e a soma de seus **n** primeiros elementos.

9. Crie valores para a largura, o comprimento e a altura de uma embalagem e calcule seu volume.

10. Crie uma variável inteira que contenha um número de segundos e imprima o número equivalente de horas, minutos e segundos.

11. Crie variáveis para a base e a altura de um retângulo e calcule sua área, perímetro e diagonal.

12. Calcule a área e o comprimento de uma circunferência de raio r = 12.

Crie as variáveis reais **a**, **b** e **c**. Calcule as raízes da equação $ax^2 + bx + c$.

Aula 3 – Estruturas condicionais e de controle

Objetivos

Utilizar corretamente as estruturas condicionais.

Verificar as diferentes estruturas de repetição.

Verificar aplicações práticas.

3.1 Comandos condicionais

Comandos condicionais são aqueles que alteram o funcionamento do programa de acordo com uma determinada condição.

Eles podem inserir interatividade entre o programa e o usuário. Existem comandos condicionais para tomada de decisões (IF-ELSE e SWITCH-CASE) e para criação de laços ou repetições (FOR, WHILE, DO-WHILE).

3.1.1 If-else

A cláusula IF (que em português significa SE) executa um bloco de instruções caso uma determinada condição seja verdadeira. A cláusula ELSE (que em português significa SENÃO) executa um bloco de instruções caso a condição seja falsa. A sintaxe do IF-ELSE no Java é a seguinte:

```
if ( <condição booleana> ) {
<código para condição verdadeira>;
}
else {
<código para condição falsa>;
}
```

Não é necessário que todo IF seja acompanhado de um ELSE, mas todo ELSE só pode existir após um IF. Uma expressão booleana é qualquer expressão que retorne *true* ou *false* e pode ser criada com os operadores de comparação e/ou lógicos.

O trecho de código a seguir, por exemplo, não imprime nada na tela, pois 37 não é maior que 40:

```
int x = 37;
if ( x > 40 ) {
System.out.println(x);
}
```

O trecho de código a seguir verifica se uma pessoa é maior de idade:

```
int x = 15;
if ( x < 18 ) {
System.out.println("Entrada permitida.");
}
else {
System.out.println("Entrada proibida.");
}
```

Podemos também concatenar expressões booleanas com os operadores lógicos "E" e "OU". O primeiro é representado por "&&", e o segundo por "||".

No exemplo a seguir, o programa verifica se uma pessoa precisa pagar passagem de acordo com sua idade (nesse caso, não pagam passagens pessoas com até 2 anos ou a partir de 60 anos):

```
if ( x > 2 && x < 60 ) {
System.out.println("Usuário deve pagar passagem.");
}
else {
System.out.println("Passagem gratuita.");
}
```

Como sugestão de *link*, utilize o endereço: http://www.youtube.com/watch?v=dijtgZiGtnA

Apesar de diferente, o trecho de código a seguir faz efetivamente o mesmo que o anterior:

```
if ( x <= 2 || x >= 60 ) {
  System.out.println("Passagem gratuita.");
}
else {
  System.out.println("Usuário deve pagar passagem.");
}
```

Criar um programa em Java para ler cinco valores inteiros e mostrar a impressão desses valores em ordem crescente.

3.1.2 Switch-case

A estrutura SWITCH-CASE equivale a um conjunto de cláusulas IF encadeadas, deixando o código mais legível e eficiente no caso de grandes desvios condicionais. Exemplo:

```
switch (x) {
case 0: System.out.println("zero"); break;
case 1: System.out.println("um"); break;
case 2: System.out.println("dois"); break;
case 3: System.out.println("tres"); break;
case 4: System.out.println("quatro"); break;
case 5: System.out.println("cinco"); break;
case 6: System.out.println("seis"); break;
case 7: System.out.println("sete"); break;
case 8: System.out.println("oito"); break;
case 9: System.out.println("nove"); break;
default : System.out.println("Número desconhecido");
}
```

3.2 Exceções em Java

O Java oferece duas importantes estruturas para o controle de erros muito semelhantes às estruturas existentes na linguagem C++: try-catch e try-finally. Ambas têm o propósito de evitar que o programador tenha que realizar testes de verificação e avaliação antes da realização de certas operações, desviando, automaticamente, o fluxo de execução para rotinas de tratamento de erro. Utilizando essas diretivas (detalhadas nas próximas seções) delimita-se um trecho de código que será monitorado, automaticamente, pelo sistema.

3.3 Uso da estrutura try catch

Quando ocorre um ou mais tipos de erros dentro de um trecho de código delimitado, o TRY-CATCH desvia, automaticamente, a execução para uma rotina designada para o tratamento específico desse erro. A sintaxe é a seguinte:

```
try {
// código normal
} catch ( <exceção 1> ) {
// código de tratamento do primeiro tipo de erro
} catch ( <exceção 2> ) {
// código de tratamento do segundo tipo de erro
} catch ( <exceção 3> ) {
// código de tratamento do terceiro tipo de erro
}
```

Por exemplo, podemos criar um programa que precisa receber um número inteiro da linha de comando. Como os argumentos são passados em um vetor de *strings*, precisamos transformar a *string* que contém o número para um inteiro. Se a conversão gerar um erro, significa que o argumento não é um número inteiro válido. A exceção usada, nesse caso, é o "java.lang. NumberFormatException".

Outro erro de que podemos tratar é o caso de não ser fornecido o argumento desse mesmo programa, utilizando a exceção "ArrayIndexOutOfBoundsException". Nesse caso, ocorrerá um erro ao tentarmos acessar o índice 0 do vetor (que está vazio). O código a seguir mostra como fazemos esses dois tratamentos com o TRY-CATCH:

```
int j = 10;
try {
  while (j > Integer.parseInt(args[0])){
            System.out.println(" "+j);
            j--;
  }
}
catch (ArrayIndexOutOfBoundsException e){
  System.out.println("Não foi fornecido um argumento.");
}
catch (java.lang.NumberFormatException e)
{
  System.out.println("Argumento não é um inteiro válido.");
}
```

Podem existir inúmeros blocos *catch* no tratamento de erros (cada um para um tipo de exceção).

3.4 Uso da estrutura try-finally

Com o TRY-FINALLY, podemos assegurar que uma rotina de finalização seja garantidamente executada mesmo que ocorra um erro (isto é, o trecho de código contido na cláusula FINALLY é executado sempre que o programa passa pela cláusula TRY). A sintaxe do TRY-FINALLY é a seguinte:

```
try {
<código normal>;
} finally {
<código que sempre deve ser executado>;
}
```

Isto é particularmente interessante quando certos recursos do sistema ou estruturas de dados devem ser liberados, independentemente de sua utilização.

Um mesmo **try** pode ser usado com as diretivas **catch** e **finally**.

A seguir, mostramos um exemplo de código utilizando TRY, CATCH e FINALLY.

```
public class TratamentoDeErro{
  public static void main(String[] args ){
        int[] array = {0, 1, 2, 3, 4, 5}; // array de 6 posições
        try{
              for(int i=0; i<10; i++ ){
                    array[i] += i;
                    System.out.println(array[i]);
              }
              System.out.println("Bloco executado com sucesso");
        }
        catch( ArrayIndexOutOfBoundsException e ){
              System.out.println("Acessou um índice inexistente»);
        }
        catch( Exception e ) {
              System.out.println(«Outro tipo de exceção ocorreu»);
        }
        finally{
              System.out.println(«Isto SEMPRE executa!»);
        }
  }
}
```

3.5 While

Utilizamos um WHILE para criarmos um laço (*loop*), ou seja, repetir um trecho de código algumas vezes enquanto uma determinada condição for verdadeira. O exemplo a seguir imprime os cinco primeiros múltiplos de 9:

```
int x = 1;
while (x <= 5) {
System.out.println(9*x);
x++;
}
```

O trecho de código dentro do WHILE será executado enquanto a condição x <= 5 for verdadeira. Isso deixará de acontecer no momento em que x > 5.

3.6 For

O comando FOR também é utilizado para criarmos *loops*. A ideia é a mesma que a do WHILE, mas existe um espaço próprio para inicializar e modificar a variável de controle do laço, deixando-o mais legível. A sintaxe do FOR é a seguinte:

```
for ( <inicialização>; <condição>; <incremento>) {
<trecho de código>;
}
```

O exemplo a seguir gera o mesmo resultado do WHILE acima:

```
for (int x = 1; x <= 5; x++) {
System.out.println(9*x);
}
```

O FOR e o WHILE podem ser usados para a mesma coisa. Porém, o código do FOR indica claramente que a variável **i** serve, em especial, para controlar a quantidade de laços executados. Use cada um quando achar mais conveniente.

O **for** também é muito útil para percorrermos um *array*. Para isso, basta usarmos o atributo "*length*", que retorna o tamanho do *array*.

Veja um exemplo:

```
for (int x = 0; x <= nome_do_array.length; x++) {
```

Exemplo0307.java

```
1  class Exemplo0307
2  {
3      public static void main (String args[])
4      {
5          for (int i=10; i>0; i--)
6          {
7              System.out.print(i + " ");
8          }
9          System.out.println(); // linha em branco
10         System.out.println("Acabou!");
11     }
12 }
```

Figura 3.1: Exemplo0307
Fonte: Elaborada pelo autor

Figura 3.2: Execução do programa Exemplo0307
Fonte: Elaborada pelo autor

Como sugestão de *link*, utilize o endereço: http://www.youtube.com/watch?v=8krfObWwZ8I&feature=related

Criar um programa em Java para mostrar os valores do fatorial dos números inteiros de 1 até 100.

Resumo

Nesta aula falamos sobre os conhecimentos necessários para a utilização correta das estruturas condicionais. Também abordamos as diferentes estruturas de repetição em Java. Citamos também como funcionam as exceções em Java.

Atividades de aprendizagem

1. Imprima o fatorial dos números de 1 a 10. Crie um *for* que comece imprimindo o fatorial de 1, e a cada passo utilize o último resultado para o cálculo do fatorial seguinte.

2. Imprima os 30 primeiros elementos da série de Fibonacci. A série é a seguinte: 1, 1, 2, 3, 5, 8, 13, 21 etc. Para calculá-la, o primeiro e segundo elementos valem 1, daí por diante, cada elemento vale a soma dos dois elementos anteriores (ex.: 8 = 5 + 3).

3. Imprima a soma de todos os números de 1 a 1.000.

4. Imprima todos os múltiplos de 3, entre 1 e 100.

5. Calcule a soma dos 70 primeiros elementos de uma Progressão Aritmética na qual o primeiro termo vale 7 e a razão vale 13.

6. Calcule a soma dos 10 primeiros elementos de uma Progressão Geométrica na qual o primeiro termo vale 3 e a razão vale 2.

7. Crie um número inteiro **n** e imprima um quadrado feito por **n** asteriscos de cada lado.

8. Imprima o fatorial de um número inteiro qualquer.

9. Crie um número inteiro qualquer e calcule a soma dos algarismos desse número.

10. Crie um número inteiro e verifique se ele é primo.

11. . Crie um número inteiro e imprima todos os seus divisores.

12. Crie uma variável com um caractere contendo uma operação ('+', '-', '*' ou '/') e outras duas com números inteiros. Execute a operação indicada pelo caractere com as duas variáveis inteiras.

13. Crie uma variável com o número de um mês e imprima o nome do mês.

14. Escreva um programa que verifique se uma nota é péssima (nota=1), ruim (2), regular (3), boa (4), ótima (5) ou nenhuma delas (nota inválida).

15. Crie três variáveis inteiras e um trecho de código que descubra a maior entre elas. Imprima as três variáveis em ordem crescente. Verifique se as mesmas três variáveis podem ser lados de um triângulo (ou seja, nenhuma pode ser maior que a soma das outras duas).

16. Crie uma variável contendo a idade de uma pessoa e verifique sua classe eleitoral: (até 16 anos não pode votar); (entre 16 e 18 anos ou mais que 65 é facultativo); (entre 18 e 65 anos é obrigatório).

17. Crie variáveis contendo as notas de três provas feitas por um aluno. Calcule a média parcial do aluno (média aritmética simples) e verifique se ele passou direto. Se não, calcule sua média final (peso 4 para a média parcial e peso 6 para uma outra variável contendo a nota de sua prova final) e verifique se ele ficou reprovado.

Aula 4 – Funções matemáticas e de *string*

Objetivos

Demonstrar as principais funções matemáticas em Java.

Demonstrar os principais métodos para manipulação de *strings* em Java.

Mostrar as técnicas de localização de caracteres em *strings*.

4.1 Funções matemáticas

A linguagem Java possui uma classe com diversos métodos especializados em realizar cálculos matemáticos. Para realizar esses cálculos, são utilizados os métodos que devem apresentar a seguinte sintaxe: Math.<nome do método> (<argumentos ou lista de argumentos>). Não é necessário importar a classe Math em um programa para poder utilizar seus recursos, pois ela já faz parte do pacote "java.lang", importado automaticamente pelo compilador do Java.

A classe Math define duas constantes matemáticas, sendo "Math.PI" – o valor de pi (p= 3,14159265358979323846) e "Math.E" que se refere ao valor da base **e** para logaritmos naturais (e=2,7182818284590452354).

A seguir, são apresentados os métodos mais comuns da classe Math.

4.1.1 Método ceil

Este método tem como função realizar o arredondamento de um número do tipo *double* para seu próximo inteiro. Sua sintaxe é a seguinte: Math.ceil (<valor do tipo double>).

No Exemplo0401 da Figura 4.1 o método **ceil** da classe math é chamado para realizar o arredondamento do número tipo *double* entre parênteses, representando nesse caso por uma variável (linhas 6 a 8). As variáveis entre parênteses compõem o argumento (do tipo *double*) do método **ceil.** Este método retorna um resultado arredondado, mantendo o tipo do dado, isto é, a variável retornada também será do tipo *double*, porém mostrando ape-

nas a parte inteira do número verificado. O tipo *double* é o único que pode ser utilizado, uma vez que o método **ceil** não aceita o tipo *float*. A Figura 4.2 apresenta a tela de resultados do Exemplo0401.

Exemplo0401.java

```
1  class Exemplo0401
2  {
3      public static void main (String args[])
4      {
5          double a = 5.2, b = 5.6, c = -5.8;
6          System.out.println("Arredondando  5.2  = " + Math.ceil(a));
7          System.out.println("Arredondando  5.6  = " + Math.ceil(b));
8          System.out.println("Arredondando -5.8  = " + Math.ceil(c));
9      }
10 }
```

Figura 4.1: Exemplo0401
Fonte: Elaborada pelo autor

Figura 4.2: Execução do programa Exemplo0401
Fonte: Elaborada pelo autor

Todos os exemplos apresentados utilizam o *prompt* (ou console) para a saída de dados.

4.1.2. Método floor

Assim como **ceil**, o método **floor** também é utilizado para arredondar um determinado número, mas para o seu inteiro anterior. Sua sintaxe é idêntica à do método **ceil**: Math.floor(<valor do tipo double>).

As Figuras 4.3 e 4.4 mostram o Exemplo0402 para ilustrar o método **floor**.

Exemplo0402.java

```
1  class Exemplo0402
2  {
3      public static void main (String args[])
4      {
5          double a = 5.2, b = 5.6, c = -5.8;
6          System.out.println("Arredondando  5.2  = " + Math.floor(a));
7          System.out.println("Arredondando  5.6  = " + Math.floor(b));
8          System.out.println("Arredondando -5.8  = " + Math.floor(c));
9      }
10 }
```

Figura 4.3: Exemplo0402
Fonte: Elaborada pelo autor

Figura 4.4: Execução do programa Exemplo0402
Fonte: Elaborada pelo autor

4.1.3 Método max

Utilizado para verificar o maior valor entre dois números, que podem ser do tipo *double*, *float*, *int* ou *long*. A sua sintaxe é a seguinte: Math.max(<valor1>,<valor2>).

Observe que o cálculo do maior número pode ocorrer entre dois números do mesmo tipo de dados ou não.

Pode-se obter o maior entre dois números do tipo *double*, entre dois números do tipo *int* ou entre um do tipo *double* e outro do tipo *int*. As Figuras 4.5 e 4.6 mostram o Exemplo0403.

Exemplo0403.java

```
1  class Exemplo0403
2  {
3    public static void main (String args[])
4    {
5      int a = 10, b = 15;
6      double c = -5.9, d = -4.5;
7      System.out.println("O maior entre 10   e 15   = " + Math.max(a,b));
8      System.out.println("O maior entre -5.9 e -4.5 = " + Math.max(c,d));
9      System.out.println("O maior entre 10   e -5.9 = " + Math.max(a,c));
10   }
11 }
```

Figura 4.5: Exemplo0403
Fonte: Elaborada pelo autor

Figura 4.6: Execução do programa exemplo0403
Fonte: Elaborada pelo autor

4.1.4 Método min

O método min fornece o resultado contrário do método **max**, sendo então utilizado para obter o valor mínimo entre dois números. Do mesmo modo que o método **max**, esses números também podem ser do tipo *double*, *float*, *int* ou *long*. A sua sintaxe é a mesma do método **max**, mudando apenas para Math.min mostrada a seguir:

Math.min(<valor1>,<valor2>)

No Exemplo0404 utiliza-se dos mesmos valores do exemplo anterior (Exemplo0403), porém troca o método **max**() pelo método **min**().

Exemplo0404.java

```
1  class Exemplo0404
2  {
3      public static void main (String args[])
4      {
5      int a = 10, b = 15;
6      double c = -5.9, d = -4.5;
7      System.out.println("O menor entre 10   e 15   = " + Math.min(a,b));
8      System.out.println("O menor entre -5.9 e -4.5 = " + Math.min(c,d));
9      System.out.println("O menor entre 10   e -5.9 = " + Math.min(a,c));
10     }
11 }
```

Figura 4.7: Exemplo0404
Fonte: Elaborada pelo autor

Figura 4.8: Execução do programa Exemplo0404
Fonte: Elaborada pelo autor

4.1.5 Método sqrt

Quando há necessidade de calcular a raiz quadrada de um determinado número, utiliza-se o método **sqrt**.

O número do qual se deseja extrair a raiz quadrada deve ser do tipo *double* e o resultado obtido também será um número do tipo*double*.

Veja sua sintaxe:

Math.sqrt(<valor do tipo Double>)

O Exemplo0405 mostra a utilização do método **sqrt** conforme Figuras 4.9 e 4.10 a seguir.

Exemplo0405.java

```
1  class Exemplo0405
2  {
3      public static void main (String args[])
4      {
5      double a = 900, b = 30.25;
6      System.out.println("A raiz quadrada de 900    = " + MT.sqrt(a));
7      System.out.println("A raiz quadrada de 30.25  = " + Math.sqrt(b));
8      }
9  }
```

Figura 4.9: Exemplo0405
Fonte: Elaborada pelo autor

Figura 4.10: Execução do programa Exemplo0405
Elaborada pelo autor

4.1.6 Método pow

Assim como é possível extrair a raiz quadrada de um número, também é possível fazer a operação inversa, ou seja, elevar um determinado número ao quadrado ou a qualquer outro valor de potência.

A potenciação de um número pode ser calculada pelo método **pow**.

Os números a serem usados no cálculo, isto é, os valores da base e da potência, devem ser do tipo *double*. Sua sintaxe é a seguinte: Math.pow(<valor da base>,<valor da potência>).

O Exemplo0406 demonstra o uso do método **pow** conforme mostrado nas Figuras 4.11 e 4.12 a seguir.

Exemplo0406.java

```
1  class Exemplo0406
2  {
3      public static void main (String args[])
4      {
5          double base = 5.5, potencia = 2;
6          System.out.println("5.5 elevado a 2   = " + Math.pow(base,potencia));
7          System.out.println("25 elevado a 0.5 = " + Math.pow(25,.5));
8          System.out.println("5678 elevado a 0 = " + Math.pow(5678,0));
9      }
10 }
```

Figura 4.11: Exemplo0406
Fonte: Elaborada pelo autor

Figura 4.12: Execução do programa Exemplo0406
Fonte: Elaborada pelo autor

4.1.7 Método random

É utilizado para gerar valores de forma aleatória. Toda vez que o método **random** é chamado, será sorteado um valor do tipo *double* entre 0 e 1 (o valor 1 nunca é sorteado). Nem sempre essa faixa de valores é suficiente numa aplicação real.

Por exemplo, para simular o sorteio de números entre 0 e 99 para um jogo de loteria qualquer, torna-se necessário o sorteio de números inteiros aleatórios no intervalo de 0 a 99.

Para que esses números possam ser sorteados, é preciso utilizar o operador de multiplicação (*) em conjunto com o método **random**. Com isso torna-se possível definir o intervalo em que o número será sorteado. O conversor (*int*) também pode ser usado para truncar a parte do ponto flutuante (a parte depois do ponto decimal) para que um número inteiro seja gerado, da seguinte forma: (int) (math.random() * 100).

Com isso seriam gerados números inteiros entre 0 e 99, atendendo plenamente á necessidade exposta.

O Exemplo0407 demonstra o uso do método **random** para simular a geração de cinco cartões de loteria com seis números cada (Figura 4.13).

Exemplo0407.java

```
1  class Exemplo0407
2  {
3      public static void main (String args[])
4      {
5          for(int qtd=1;qtd<=5;qtd++)
6          {
7              for(int x=1;x<=6;x++)
8              {
9                  int num = (int) (Math.random() * 100);
10                 System.out.print(num + "  ");
11             }
12             System.out.println();
13         }
14     }
15 }
```

Figura 4.13: Exemplo0407
Fonte: Elaborada pelo autor

Figura 4.14: Execução do programa Exemplo0407
Fonte: Elaborada pelo autor

No Exemplo0407 o primeiro *loop for* (linha 5) é o responsável pela contagem de um a cinco (os cinco cartões). O segundo *loop for* (linha 7) é o responsável pela contagem de um a seis (os seis números de cada cartão). Os números são mostrados um ao lado do outro e a cada cartão é pulada uma linha em branco (linha 12). A Figura 4.14 mostra o resultado da execução do programa Exemplo0407.

4.1.8 Formatação com a classe DecimalFormat

Os cálculos matemáticos, em especial os que envolvem multiplicação e divisão, podem gerar resultados com muitas casas decimais. Isso nem sempre é necessário e esteticamente correto, pois apresentar um resultado com muitas casas decimais não é muito agradável e legível à maioria dos usuários. Por exemplo: considere duas variáveis do tipo *double* x=1 e y=6. Ao realizar a divisão de x por y, aparece na tela o resultado 0,166666666666.

Esse resultado não é o mais adequado para se apresentar na tela. Seria mais conveniente mostrar o resultado formatado com duas ou três casas decimais.

Para realizar a formatação, é necessário definir um modelo conhecido pelo nome de **pattern**.

Considere **pattern** como o estilo de formatação que será apresentado sobre um valor numérico.

Em outras palavras, você terá de informar ao compilador qual estilo de formatação deve ser usado para apresentar um número.

Para definir o **pattern**, são usados caracteres especiais. O Quadro 4.1 apresenta os caracteres mais usados.

Quadro 4.1: Caracteres mais utilizados na classe **DecimalFormat**

Caractere	Significado
0	Imprime o dígito normalmente, ou caso ele não exista, coloca 0 em seu lugar. Exemplo: sejam as variáveis *int* x=4, y=32 e z=154, ao usar o **pattern** "000", o resultado impresso na tela seria x - 004,y - 032 e z - 154.
#	Imprime o dígito normalmente, desprezando os zeros à esquerda do número. Exemplo: sejam as variáveis *double* x=0.4 e y= 01.34, ao usar o **pattern** " ##.##", o resultado impresso na tela seria x - . 4,y - 1.34.
.	Separador decimal ou separador decimal monetário (depende do sistema usado).
-	Sinal de número negativo. Para realizar a formatação de números, vamos usar a classe **DecimalFormat**.

Para realizar a formatação de números, vamos usar a classe **DecimalFormat** conforme ilustrado no Exemplo0408 nas Figuras 4.15 e 4.16 a seguir.

Exemplo0408.java

```
1  import java.text.DecimalFormat;
2  class Exemplo0408
3  {
4      public static void main(String args[])
5      {
6          DecimalFormat  df = new DecimalFormat();
7          short idade = 38;
8          df.applyPattern("000");
9          System.out.println("Idade       : " + df.format(idade));
10         int quantidade = 9750;
11         df.applyPattern("#0,000");
12         System.out.println("Quantidade  : " + df.format(quantidade));
13         long estoque = 198564;
14         df.applyPattern("#,##0,000");
15         System.out.println("Estoque     : " + df.format(estoque));
16         float altura = 1.74f;
17         df.applyPattern("#0.00");
18         System.out.println("altura      : " + df.format(altura));
19           double peso = 70.25;
20         df.applyPattern("#0.00");
21         System.out.println("Peso        : " + df.format(peso));
22         String valorEmReais = "2583.75";
23         df.applyPattern("R$ #,##0.00");
24         System.out.println("Valor em Reais: " + df.format(Double.parseDouble(
           valorEmReais)));
25     }
26 }
```

Figura 4.15: Exemplo0408
Fonte: Elaborada pelo autor

Figura 4.16: Execução do programa Exemplo0408
Fonte: Elaborada pelo autor

Funcionalidades mais importantes no Exemplo0408:

- Linha 1: importa a classe **DecimalFormat** do pacote java.text, uma vez que ela não pertence ao conjunto de classes *default* do pacote java.lang.

- Linha 6: declara um objeto (df) de classe **DecimalFormat** que será usado para realizar a formatação dos números pelo método **format** (df.format). Essa linha poderia conter a definição do **pattern** no momento da inicialização do objeto df. Uma definição válida pode ser: **DecimalFformat** df = decimalformat ("000"). É possível observar que a definição do **pattern** pode ser realizada dentro dos parênteses.

- Linha 8: contém a definição do *pattern* pelo método applypattern (df.appypattern('000"). Essa instrução define que todos os números impressos a partir do objeto df serão formatados com três dígitos, mesmo que eles possuam menos que isso, conforme exemplificado no Quadro 4.1. As linhas 12,16, 20, 24 e 28 redefinem o **pattern**,

aplicando novas formatações ao objeto df, isto é, aos números que serão impressos pelo método **format**.

- Linha 29: apresenta uma maneira de formatar um número a partir de um formato *string*. Observe que a variável **valorEmReais** armazena um conteúdo do tipo *string*, que não pode ser manipulado diretamente pela classe **DecimalFormat**. Para que isso seja possível, o valor *string* é convertido no tipo *double* pelo método **parseDouble** da classe *double* (Double.parseDouble (valorEmReais).

O resultado da execução do programa Exemplo0408 aparece na Figura 4.16

4.2 Funções com *strings*

Uma *string* é um tipo que corresponde à união de um conjunto de caracteres.

Em Java, as *strings* são instâncias da classe **String**, isto é, geram objetos que possuem propriedades e métodos, diferentemente dos tipos primitivos com *int*, *float*, *double*, etc.

Essas *strings* podem ser manipuladas de várias formas. Por exemplo, é possível verificar seu comprimento, retirar um pedaço dela, acessar ou mudar caracteres individuais. As *strings* constituem uma cadeia de caracteres entre aspas. Exemplo: Frase = "linguagem Java" da mesma forma que as funções matemáticas, existem diversos métodos para manipulação de *strings*, os quais acompanham a seguinte sintaxe:

<Nome da string>.<nome-do-metodo>(argumentos>)

A seguir, são apresentados os métodos mais comuns (e mais usados) da classe **String**.

4.2.1 Método length

O método **length** é muito utilizado para retornar o tamanho de uma determinada *string*, incluindo também os espaços em branco que estão presentes. Esse método retorna sempre um valor do tipo *int*.

Na prática, o método **length** é muito utilizado quando é necessário ler uma variável *string* do começo até o final, tanto para a busca de caracteres ou palavras quanto para a criação de *banners*, algo extremamente usado na internet.

Veja sua sintaxe: < string>.length()

O Exemplo0410 mostra o uso do método **length** conforme Figuras 4.17 e 4.18 a seguir.

```
Exemplo0410.java
1  class Exemplo0410
2  {
3      public static void main (String args[])
4      {
5          string frase = "Aprendendo Java";
6          int tamanho;
7          tamanho = frase.length();
8          System.out.println("String: " + frase);
9          System.out.println("Tamanho da string = " + tamanho);
10     }
11 }
```

Figura 4.17: Exemplo0410
Fonte: Elaborada pelo autor

Figura 4.18: Execução do programa Exemplo0410
Fonte: Elaborada pelo autor

No Exemplo0410, a linha 5 contém a declaração da *string* frase. Cabe uma observação: conforme citado anteriormente, na realidade frase não é uma variável e sim um objeto, pois uma variável não pode conter métodos atrelados a ela; somente os objetos que possuem métodos para manipulação de suas informações. A linha 7 contém a utilização de **length** por meio de "frase.length()", isto é, retorna o número de caracteres armazenado em frase (no caso 15).

Em vez de usar "frase.length()" poderia ser utilizada a forma literal do seguinte modo: tamanho="Aprendendo Java" . length(). O resultado seria o mesmo.

4.2.2 Método charAt

Usando para retornar um caractere de determinada *string* de acordo com um índice especificado entre parênteses. Esse índice refere-se à posição do caractere na *string*, sendo 0 (zero) o índice do primeiro caractere, 1 (um) o do segundo e assim por diante.

O método **charAt** é útil quando for necessário verificar a existência de um caractere na *string*.

Por exemplo: suponha que uma determinada *string* só possa conter números – a função **charAt** pode ser usada para verificar a existência de dígitos numéricos nessa *string*.

A sintaxe do método **charAt** é a seguinte: < string>.charAt (<índice>)

O Exemplo0411 mostra o uso do método **charAt** conforme as Figuras 4.19 e 4.20 a seguir.

Exemplo0411.java

```
1  class Exemplo0411
2  {
3      public static void main (string args[])
4      {
5          String frase = "Aprendendo Java";
6          System.out.println("String   = " + frase);
7          System.out.println("Caracter = " + frase.charAt(5));
8          System.out.print("Trecho   = ");
9          for (int i=11;i<=14;i++)
10         {
11             System.out.print(frase.charAt(i));
12         }
13         System.out.println();
14     }
15 }
```

Figura 4.19: Exemplo0411
Fonte: Elaborada pelo autor

```
C:\Windows\system32\cmd.exe
D:\Cap4>javac Exemplo0411.java

D:\Cap4>java Exemplo0411
String   = Aprendendo Java
Caracter = d
Trecho   = Java

D:\Cap4>_
```

Figura 4.20: Execução do programa Exemplo0411
Fonte: Elaborada pelo autor

Veja na Figura 4.20, os resultados desse exemplo e repare que o quinto caractere apresentado é o "d" e não "n", pois o índice começa a partir do zero. As linhas 9 a 12 são responsáveis por apresentar o trecho equivalente aos índices de número 11 a 14, cujos caracteres correspondem à palavra Java armazenada na variável **frase**.

4.2.3 Métodos toUpperCase e toLowerCase

Os métodos **toUpperCase** e **toLowerCase** são utilizados para transformar todas as letras de uma determinada *string* em maiúsculas ou minúsculas. O método **toUpperCase** transforma todos os caracteres de uma *string* em maiúsculos. O método **toLowerCase** transforma todos os caracteres de uma *string* em minúsculos. Sua sintaxe é a seguinte: < string>.toUpperCase() ou <String>.toLowerCase().

O Exemplo0412 demonstra o uso dos métodos **toUpperCase** e **toLowerCase** e dispensa mais detalhes, dada a simplicidade dessas duas funções. A

única observação se refere ao fato de que esses métodos não alteram o valor original da *string*. Mesmo aplicando os métodos das linhas 6 e 7, o conteúdo das variáveis **palavra1** e **palavra2** permanece o mesmo, isto é, a transformação ocorre apenas com fins de impressão em tela.

Se for necessário alterar o conteúdo de uma variável *string*, substituindo seu valor original pelo transformado, a própria variável deve receber o valor de sua transformação, por exemplo: palavra1=palavra.toLowerCase().

Exemplo0412.java

```
1  class Exemplo0412
2  {
3      public static void main (String args[])
4      {
5          String palavra = "LINGUAGEM JAVA";
6          System.out.println(palavra.substring(10,14));
7          String palavra1 = "ARROZ", palavra2 = "batata";
8          System.out.println("ARROZ em min£scula  = " + palavra1.toLowerCase());
9          System.out.println("batata em mai£scula = " + palavra2.toUpperCase());
10         System.out.println("SaLaDa em min£scula = " + "SaLaDa".toLowerCase());
11     }
12 }
```

Figura 4.21: Exemplo0412
Elaborada pelo autor

Figura 4.22: Execução do programa Exemplo0412
Fonte: Elaborada pelo autor

4.2.4 Método substring

Ele retorna a cópia de caracteres de uma *string* a partir de dois índices inteiros especificados, funcionando basicamente da mesma forma que o método **charAt**, dentro de um *lopping*, conforme indica a seção 4.2.2. A sintaxe de *substring* é a seguinte: < string>.*substring*(< índice inicial>,[<indice final>]).

O primeiro argumento especifica o índice a partir do qual se inicia a cópia dos caracteres (da mesma forma que **charAt**, o índice inicia-se em 0). O segundo argumento é opcional e especifica o índice final, em que termina a cópia dos caracteres; entretanto, o índice final deve especificar um índice além do último caractere.

Para melhor entendimento do método **substring,** considere a variável **frase** com o seguinte conteúdo:

Frase	L	I	N	G	U	A	G	E	M		J	A	V	A
Índice	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Cada caractere de uma variável *string* é indexado a partir do 0 (zero). Vamos apresentar alguns exemplos:

1. String x= frase. substring (10) - x Recebe o conteúdo "JAVA" , pois ao passar apenas o primeiro argumento para o método **substring**, ele retorna da posição informada (no caso 10, a posição da letra J) até o último caractere da *string*.

2. String x= frase. substring (3) - x Recebe o conteúdo "GUAGEM JAVA" isto é, do caractere de índice 3 até o último caractere da *string* frase.

3. String x= frase. substring (3,9) - x Recebe o conteúdo "GUAGEM" , isto é, do caractere de índice 3 até o caractere de índice 8 (9-1).

4. String x= frase. substring (0,1) - x Recebe o conteúdo "L" , isto é, do caractere de índice 0 até o caractere de índice 0 (1-1).

5. String x= frase. substring (10,14) - x Recebe o conteúdo "JAVA" , isto é, do caractere de índice 10 até o caractere de índice 13 (14-1). Observe que o resultado deste exemplo é igual ao do exemplo 1.

Se os índices especificados estiverem fora dos limites da *string*, é gerado o erro "stringIndexOutOfBoundsException". No exemplo, se você usar "frase. substring(10,20)" ocorre o erro citado, uma vez que não existe índice 20.

O Exemplo0413 apresenta um código que usa o método **substring** para separar as palavras de uma frase pela manipulação de seus índices, ilustrado nas Figuras 4.23 e 4.24 a seguir.

```java
class Exemplo0413
{
    public static void main (String args[])
    {
        String frase = "Eu gosto de java!";
        System.out.println("Separando uma frase em palavras:");
        System.out.println("Palavra 1: " + frase.substring(0,2));
        System.out.println("Palavra 2: " + frase.substring(3,8));
        System.out.println("Palavra 3: " + frase.substring(9,11));
        System.out.println("Palavra 4: " + frase.substring(12));
    }
}
```

Figura 4.23: Exemplo0413
Fonte: Elaborada pelo autor

```
C:\Windows\system32\cmd.exe
D:\Cap4>java Exemplo0413
Separando uma frase em palavras:
Palavra 1: Eu
Palavra 2: gosto
Palavra 3: de
Palavra 4: java!

D:\Cap4>_
```

Figura 4.24: Execução do programa Exemplo0413
Fonte: Elaborada pelo autor

4.2.5 Método trim

Seu objetivo é remover todos os espaços em branco que aparecem no início e no final de uma determinada *string*. São removidos apenas os espaços do início e do fim da *string*.

Não são removidos os espaços entre as palavras.

Sua sintaxe é a seguinte:

<string>.trim()

O Exemplo0414 mostra a utilização do método **trim** conforme as Figuras 4.25 e 4.26 a seguir.

```java
class Exemplo0414
{
    public static void main (String args[])
    {
        String frase = "  Jesus Cristo: o rei dos reis.  ";
        System.out.println("Com espacos: " + "*" + frase + "*");
        System.out.println("Sem espacos: " + "*" + frase.trim() + "*");
    }
}
```

Figura 4.25: Exemplo0414
Fonte: Elaborada pelo autor

Figura 4.26: Execução do programa Exemplo0414
Fonte: Elaborada pelo autor

A variável frase armazena uma *string* com espaços em branco no início e no final. Quando não é utilizado o método **trim**, os espaços permanecem na *string*: em contrapartida, ao usar **trim**, os espaços desaparecem.

O método **frase. trim ()** não retirou, realmente, os espaços em branco da variável, ou seja, apenas foi mostrado na tela um **trim** da variável, que consequentemente a exibe sem os espaços em branco, mas a variável em si ainda continua com os espaços no seu início e no seu final.

Para que realmente os espaços sejam retirados, é necessário que o resultado de **trim** seja atribuído à própria variável, com a seguinte instrução: frase=frase.trim(). Isso tornaria a variável livre dos espaços em branco.

4.2.6 Método replace

É utilizado para substituição de caracteres, ou grupo de caracteres, em uma determinada *string*. Para seu funcionamento, é necessário informar o(s) caractere(s) que deseja substituir e por qual(is) caractere(s) ele(s) será(ão) substituídos(s). Caso não haja na *string* nenhuma ocorrência do caractere a ser substituído, a *string* original é retornada, isto é, não ocorre nenhuma alteração.

No Exemplo0415, a linha 5 declara uma *string* (frase1) que recebe uma frase. A linha 6 armazena essa frase na variável **frase2**, porém sem os espaços em branco, uma vez que o método **replace** foi usado para substituir todos os espaços por vazios (‘ ‘ por “”).

As linhas 8 e 9 substituem um caractere por outro (“ “ por “_” e “a” por “u”) e a linha 10 substitui uma palavra por outra (“na” por “NA”). Da mesma forma que **trim**, o método **replace** não altera o conteúdo da variável. Para fazer com que uma variável receba o resultado de uma troca de caracteres, faça como apresentado na linha 6.

Para melhor compreensão dos resultados observe o Exemplo0415 nas Figuras 4.27 e 4.28 a seguir.

```java
class Exemplo0415
{
    public static void main (String args[])
    {
        String frase1 = "o macaco gosta de banana";
        String frase2 = frase1.replace(" ","");
        System.out.println("Frase 1: " + frase1);
        System.out.println("Frase 2: " + frase2);
        System.out.println("Frase 3: " + frase1.replace(" ", "_"));
        System.out.println("Frase 4: " + frase1.replace("a", "u"));
        System.out.println("Frase 5: " + frase1.replace("na", "NA"));
    }
}
```

Figura 4.27: Exemplo0415
Fonte: Elaborada pelo autor

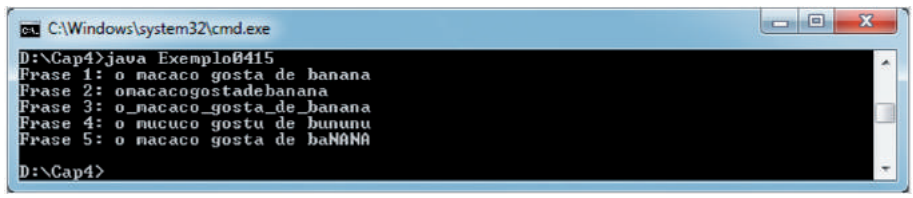


Figura 4.28: Execução do programa Exemplo0415
Fonte: Elaborada pelo autor

4.2.7 Método valueOf

O método **valueOf** é usado para converter diversos tipos de dados em *strings*. Esse método aceita vários tipos de argumento (números ou cadeia de caracteres) e transforma-os em *strings*. Esta seção aborda apenas a conversão de tipos numéricos em *strings*. Uma das sintaxes possíveis para o método **valueOf** é :

string.valueOf(<nome da variável a ser convertida>)

Para facilitar o entendimento, o Exemplo0416 demonstra a conversão de vários tipos numéricos com o uso do método **valueOf** conforme as Figuras 4.29 e 4.30 a seguir.

```java
class Exemplo0416
{
    public static void main (String args[])
    {
        String x = "";
        int a = 11;
        long b = 222;
        float c = 3333;
        double d = 4.444;
        x = x + String.valueOf(a) + " - ";
        x = x + String.valueOf(b) + " - ";
        x = x + String.valueOf(c) + " - ";
        x = x + String.valueOf(d);
        System.out.println("Valores convertidos:");
        System.out.println(x);
    }
}
```

Figura 4.29: Exemplo0416
Fonte: Elaborada pelo autor

No Exemplo0416 todas as variáveis numéricas (a,b,c,d) declaradas nas linhas 6 a 9 são convertidas e acumuladas em uma variável *string* (x) nas linhas 10 a 13. Essa não é a funcionalidade mais importantes do método **valueOf**, uma vez que o mesmo resultado pode ser alcançado sem sua utilização por meio da concatenação das variáveis com o operador de concatenação (+), conforme demonstrado em seguida:

Figura 4.30: Execução do programa Exemplo0416
Fonte: Elaborada pelo autor

4.2.8 Método indexOf

O método **indexOf** é usado para localizar caracteres ou *substrings* em uma *string*. Quando realizamos a busca de uma palavra em um texto, usamos algo parecido com o funcionamento de **indexOf**, isto é, ele busca uma palavra e retorna a posição onde ela se encontra.

Você já sabe que um texto (ou uma *string*) é indexado a partir do número zero.

Caso haja sucesso na busca, é retornado um número inteiro referente à posição do texto (o índice) onde o caractere foi encontrado, ou a posição do texto onde inicia a *substring* localizada. Caso haja insucesso na busca, isto é, caso o caractere ou *substring* não tenha sido encontrado, é retornado o

valor inteiro -1. De qualquer modo, o retorno de **indexOf** sempre será um número inteiro (o valor do índice, ou -1). A sintaxe geral para utilização do método **indexOf** é:

string.indexof(<caractere ou substring a ser localizada, [posição inicial]>)

No Exemplo0417 (Figuras 4.31 e 4.32) verificamos que a linha 5 contém o texto que será usado nas pesquisas. A linha 6 declara um caractere 'a' que será buscado no texto. As formas de busca são as seguintes:

Exemplo0417.java

```
1  class Exemplo0417
2  {
3      public static void main (String args[])
4      {
5          String texto = "Livro Java 2 - Ensino didatico";
6          char caractere = 'a';
7          System.out.println("indice: " + texto.indexOf(caractere));
8          System.out.println("indice: " + texto.indexOf(caractere,10));
9          System.out.println("indice: " + texto.indexOf("Ensino"));
10         System.out.println("indice: " + texto.indexOf("Java",15));
11     }
12 }
```

Figura 4.31: Exemplo0417
Fonte: Elaborada pelo autor

- linha 7: busca o caractere 'a' no texto e retorna o índice referente à sua primeira ocorrência. No caso, retorna o índice 7.
- linha 8: busca o caractere 'a' no texto a partir do índice 10 e retorna o índice referente à sua primeira ocorrência. No caso, retorna o índice 25. Na realidade, a primeira ocorrência do caractere 'a' seria na posição 7; entretanto, foi solicitado que a busca iniciasse na posição 10.
- linha 9: busca a substring "Ensino" no texto e retorna o índice referente à sua primeira ocorrência. No caso, retorna o índice 15.
- linha 10: busca a substring "Java" no texto a partir da posição 15 e retorna o índice referente e à sua primeira ocorrência. Como não existe a palavra "Java" após a posição 15, é retornado o valor -1 (busca sem sucesso). O mesmo princípio é aplicado quando você procura uma palavra em um editor de textos e ele não a encontra.

Figura 4.32: Execução do programa Exemplo0417
Fonte: Elaborada pelo autor

Resumo

Nesta aula falamos sobre os conhecimentos necessários para a utilização correta das funções matemáticas e de *strings*. Aprendemos a trabalhar com pesquisa de *substrings* e determinar o comprimento de uma *string*.

Atividades de aprendizagem

1. Crie uma classe que simule a jogada de um dado de seis lados dez vezes e mostre o resultado na tela.

2. Crie uma classe que calcule quantos metros cúbicos de água suporta uma determinada caixa de água em forma de cubo – todos os lados são iguais. O usuário deve informar o valor do lado e o volume será calculado pela fórmula volume=lado3. Arredonde o valor para seu inteiro anterior.

3. Construa uma classe que receba uma frase qualquer e mostre-a de forma invertida.

4. Elabore uma classe que mostre o efeito:

Frase: Java

Efeito

J

Ja

Jav

Java

Jav

Ja

J

Aula 5 – Criando funções

Objetivos

Identificar os principais tipos de métodos em Java.

Introduzir o conceito de modularidade.

Mostrar as técnicas de criação de métodos em Java.

5.1 Criação de métodos em Java

Métodos são trechos de programa que permitem modularizar um sistema, isto é, são pequenos blocos que, juntos, compõem um sistema maior. Os métodos recebem um determinado nome e podem ser chamadas várias vezes durante a execução de uma classe.

A-Z

Método
É uma sub-rotina que pode ser invocada toda vez que sua funcionalidade for necessária em um trecho da classe ou ainda a partir de outra classe.

Os principais motivos que levam à utilização de métodos se referem à redução do código de um sistema, à melhoria da modularização do sistema e à facilitação da manutenção do sistema.

Para ilustrar esses conceitos, imagine um grande sistema envolvendo muitas classes em Java, que existe a necessidade de verificação se uma determinada data é válida. Imagine, ainda, que há diversas aplicações onde isso deve ser realizado.

Um método pode invocar outro método, isto é, durante a execução do método 1 pode ser necessária a execução do método 2, que pode invocar o método 3, e assim por diante. Todo método possui uma declaração e um corpo cuja estrutura é declarada a seguir:

```
Qualificador tipo_retorno_metodo nome_metodo ([lista de argumentos])
{
        Código do corpo;
}
```

Qualificador
É conhecido também pelo nome de modificador e define a visibilidade do método. Trata-se de uma forma de especificar se o método é visível apenas para a própria classe em que está declarada, ou pode ser visualizado e utilizado por classes externas.

O **qualificador** pode ser do tipo:

a) *Public*: o método é visível por qualquer classe. É o qualificador mais aberto no sentido de que qualquer classe pode usar esse método.

b) *Private*: o método é visível apenas pela própria classe. É o qualificador mais restritivo.

c) *Protected*: o método é visível pela própria classe, por suas subclasses e pelas classes do mesmo pacote.

Tipo de retorno: refere-se ao tipo de dado retornado pelo método. Métodos que não retornam valores devem possuir nesse parâmetro a palavra *void*. Sempre que *void* é utilizada em uma declaração de método, nenhum valor é retornado após sua execução.

Nome do método: pode ser qualquer palavra ou frase, desde que iniciada por uma letra. Se o nome for uma frase, não pode conter espaços em branco. Por padrão, todo nome de método inicia com letra maiúscula.

Lista de argumentos: trata-se de uma lista de valores opcionais, que podem ser recebidos pelo método de tratamento interno. Quando um método é invocado ele pode receber valores de quem o chamou. Esses valores podem ser manipulados internamente e devolvidos ao emissor da solicitação.

Código do corpo: trata-se dos códigos em Java que realizam os processos internos e retornam os valores desejados, isto é, constituem o programa do método.

5.2 Métodos sem retorno

Não retornam valores e são semelhantes às *procedures* encontradas na maioria das linguagens de programação.

Os métodos que não retornam valores devem ser definidos como *void*.

Dessa forma, todos os métodos sem retorno observam a seguinte estrutura:

```
Qualificador void nome_método ([lista de argumentos])
{
        Código do corpo;
}
```

Algumas declarações possíveis:

- Public void imprime()
- Public static void imprime()
- Private void imprimeteste()
- Protected void gravatexto()

Vamos trabalhar com a prática e mostrar um exemplo de método em que ele é chamado para que uma mensagem seja mostrada na tela. O Exemplo0501 mostra a chamada de um método que imprime na tela uma frase qualquer, conforme ilustrado nas Figuras 5.1 e 5.2 a seguir.

Exemplo0501.java

```
1  class Exemplo0501
2  {
3      public static void main (String args[])
4      {
5          imprime(); // invocacao do metodo
6      }
7
8      private static void imprime() // declaracao do metodo
9      {
10         System.out.println("Aprendendo a Linguagem Java");
11     }
12 }
```

Figura 5.1: Exemplo0501
Elaborada pelo autor

Figura 5.2: Execução do programa Exemplo0501
Fonte: Elaborada pelo autor

A classe Exemplo0501 possui dois métodos: **main()** e **imprime()**.

Observe que cada método possui um início e um encerramento por meio de abre e fecha chaves. Toda vez que um método for declarado, ele deve obrigatoriamente possuir uma chave inicial e uma final.

A chamada de um método deve corresponder exatamente à sua declaração, ou melhor, à sua assinatura. Quando invocado, o método deve possuir o mesmo tipo de retorno, o mesmo nome e o mesmo número de argumentos.

Quando declarada a palavra reservada *void*, significa que não existe um valor de retorno. O método declarado como **public**, como já foi falado, possibilita que ele seja utilizado externamente à classe que é declarada.

O Exemplo0502 demonstra o uso de um método com passagem de argumentos, conforme ilustrado nas Figuras 5.3 e 5.4 a seguir. Neste exemplo utiliza-se um método para imprimir o conteúdo de uma variável *string* qualquer, funcionando de forma similar a "System.out.println()".

Exemplo0502.java

```
1  class Exemplo0502
2  {
3      public static void main (String args[])
4      {
5          String frase1 = "Sou muito";
6          String frase2 = "feliz porque sei";
7          String frase3 = "que Deus me ama!";
8          limpaTela();
9          tela(frase1); // invocacao do metodo
10         tela(" ");
11         tela(frase2);
12         tela(" ");
13         tela(frase3);
14     }
15     public static void tela(String texto) // declaracao do metodo
16     {
17         System.out.println(texto);
18     }
19
20     public static void limpaTela() // declaracao do metodo
21     {
22         for(int i=1;i<=25;i++)
23         {
24           System.out.println();
25         }
26     }
27 }
```

Figura 5.3: Exemplo0502
Fonte: Elaborada pelo autor

Figura 5.4: Execução do programa Exemplo0502
Fonte: Elaborada pelo autor

O Exemplo0502 é bem parecido com o anterior, só que as variáveis (**frase1**, **frase2**, **frase3**) recebem uma *string* que é passada para o método **tela()** que se encarrega da sua impressão na tela.

A classe possui três métodos: o método **main**, obrigatório em uma classe executável, o método **tela** que imprime uma frase na tela e o método **limpatela** que realiza a limpeza da tela.

O nome usado para invocar o método tela que recebe o argumento é "tela(fraseN)", no qual:

Tela: Nome do método;

FraseN: É o conteúdo que será enviado ao método, ou seja, o método é invocado e recebe uma variável do tipo *string*.

Ao declarar o método **public static void tela**(*string* texto), foi informado, entre parênteses, o tipo de variável a ser recebido(*string* texto).

O método **tela** será invocado cinco vezes pelo método **main**.

O método **limpatela** vai imprimir 25 linhas em branco na tela.

O Exemplo0503 apresenta outra classe com dois métodos que não retornam valores (além do **main**) e são executados em cascata, uma vez que o método **main** chamará **tela**, que chamará **tempo** conforme ilustrado nas Figuras 5.5 e 5.6 a seguir.

Exemplo0503.java

```
class Exemplo0503
{
    public static void main (String args[])
    {
        String frase1 = "Sou muito";
        String frase2 = "feliz porque sei";
        String frase3 = "que Deus me ama!";
        tela(frase1); // invocacao do metodo
        tela(frase2);
        tela(frase3);
    }

    public static void tela(String texto) // declaracao do metodo texto
    {
        System.out.println(texto);
        tempo(1);
    }

    public static void tempo(int segundos) // declaracao do metodo segundos
    {
        try
        {
          Thread t;
          t = new Thread();
          t.start();
          t.sleep(segundos * 1000);
        }
        catch (InterruptedException e) { }
    }
}
```

Figura 5.5: Exemplo0503
Fonte: Elaborada pelo autor

Figura 5.6: Execução do programa Exemplo0503
Fonte: Elaborada pelo autor

Neste exemplo foi incluído um novo método, chamado **tempo**, que funciona como temporizador durante a execução do programa.

5.3 Métodos com retorno de valores

A sintaxe para a declaração de métodos que retornam valores é a mesma apresentada anteriormente.

```
Public static int soma(int x, int y)
```

A declaração desse método informa que ele receberá dois argumentos inteiros (x,y) e retornará um número do tipo inteiro (*int*).

Os valores recebidos e retornados não precisam ser, necessariamente, do mesmo tipo, conforme aparece na declaração.

Podem existir métodos que recebem números e retornam uma *string*, recebem inteiros e retornam números com ponto flutuante ou qualquer outra combinação.

O Exemplo0504 apresentado nas Figuras 5.7 e 5.8 demonstram a utilização de um método que recebe duas variáveis do tipo *string* e retorna a soma entre elas na forma de um número do tipo inteiro.

```java
import javax.swing.*;
class Exemplo0504
{
    public static void main (String args[])
    {
        String n1 = JOptionPane.showInputDialog(null, "forneça o 1º número inteiro");
        String n2 = JOptionPane.showInputDialog(null, "forneça o 2º número inteiro");
        int res = soma(n1,n2);
        JOptionPane.showMessageDialog(null, "Numeros fornecidos : " + n1 + ", "+ n2 +
          "\nResultado = " + res);
    }

    public static int soma(String num1, String num2) // declaracao do metodo
    {
        int x = 0,y = 0;
        try
        {
          x = Integer.parseInt(num1);
          y = Integer.parseInt(num2);
        }
        catch(NumberFormatException e)
        {
          JOptionPane.showMessageDialog(null, "Digite apenas caracteres numericos!");
          System.exit(0); // caso houver erro encerra o programa
        }
        return (x + y); // retorna a soma dos argumentos passados
    }
}
```

Figura 5.7: Exemplo0504
Fonte: Elaborada pelo autor

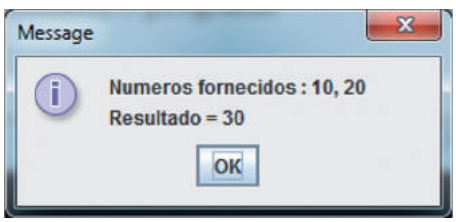


Figura 5.8: Execução do programa Exemplo0504
Fonte: Elaborada pelo autor

Ao executar a aplicação, o método **main** é executado e invoca a execução do método **soma**. O resultado da execução do método **soma** é armazenado na variável **res**, cujo tipo é o mesmo declarado para o retorno do método. Ao invocar o método **soma**, são enviadas duas variáveis do tipo *string* que se referem aos valores fornecidos pelo usuário no momento da execução do programa (n1, n2).

Ao ser invocado, o método **soma** recebe duas variáveis do tipo *string*, **num1** que recebe o conteúdo de **n1** e **num2** que recebe o conteúdo de **n2**, executa suas tarefas internas entre chaves e retorna um valor inteiro por meio da palavra *return*.

Todo método que não foi declarado como *void*, isto é, que retornar algum valor, necessita obrigatoriamente utilizar o método **return()** para retornar um valor. O valor retornado deve ser sempre do mesmo tipo declarado no método.

5.4 Recursividade

Os programas são geralmente estruturados como métodos que chamam uns aos outros, o que facilita a resolução de muitos problemas, além de reduzir consideravelmente o tamanho do código.

A recursividade ocorre quando um método chama a si próprio, direta ou indiretamente, por meio de outro método.

Para entender corretamente o funcionamento desse tipo de método, é necessário muita dedicação por parte dos desenvolvedores, pois sua utilização pode ser complexa.

O Exemplo0505 mostra uma recursividade gerada para imprimir o fatorial dos números inteiros de 0 a 10, conforme ilustrado nas Figuras 5.9 e 5.10 a seguir. Observe neste exemplo existe uma chamada ao próprio fatorial, isto é, ele chama a si mesmo. A cada vez que o método é chamado, o valor da variável *num* é diminuído de 1.

Exemplo0505.java

```
1  class Exemplo0505
2  {
3      public static void main (String args[])
4      {
5          for(long i=0; i<=10;i++)
6          {
7              System.out.println(i + "! = " + fatorial(i));
8          }
9      }
10
11     public static long fatorial(long num)
12     {
13         if (num <= 1)
14         {
15             return (1);
16         }
17         else
18         {
19             return (num * fatorial(num - 1));
20         }
21     }
22 }
```

Figura 5.9: Exemplo0505
Fonte: Elaborada pelo autor

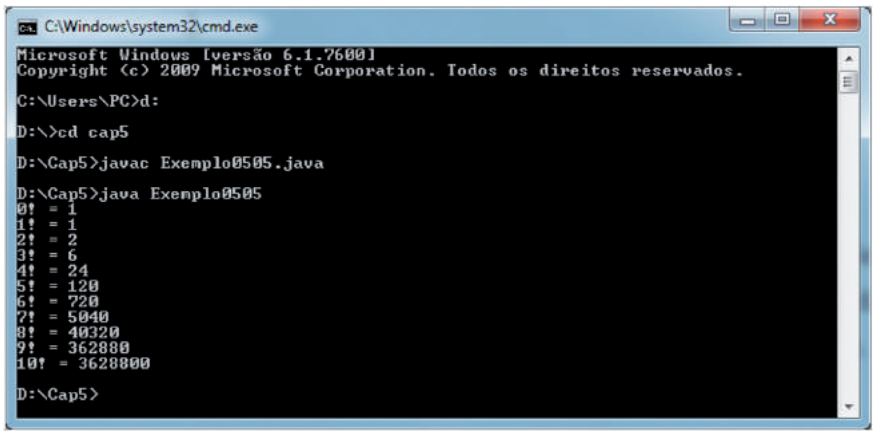


Figura 5.10: Execução do programa Exemplo0505
Fonte: Elaborada pelo autor

Resumo

Nesta aula falamos sobre os principais métodos existentes em Java, desde os que retornam até os que não retornam valores, o acesso a métodos e a recursividade. Verificamos a importância de trabalhar com recursividade na linguagem Java pela execução de exemplos. Trabalhamos com exemplos de métodos em Java com retorno de *strings* e inteiros.

Atividades de aprendizagem

1. Crie uma classe que tenha um método o qual desenhe uma moldura na tela. Essa moldura deve ter 80 caracteres de comprimento por 5 de largura. Para isso utilize a sequência de caracteres ASC II.

2. Elabore uma classe que receba o raio de uma esfera do tipo *double* e chame o método **volume_esfera** para calcular e exibir o volume da esfera da tela. A fórmula a ser utilizada é (4/3)* PI * raio * raio.

3. Construa uma classe que receba uma temperatura qualquer em Fahrenheit e apresente seu valor correspondente em Celsius por um método. Para calcular utilize a fórmula: =5/9 *(f-32).

Aula 6 – Utilizando vetores e matrizes

Objetivos

Diferenciar vetores e matrizes.

Demonstrar a praticidade de utilização de vetores.

Apresentar as vantagens de usar *arrays*.

6.1 Definição de *array*

Em determinados rotinas de programa torna-se necessário manipular diversas variáveis de um mesmo tipo de dado, por exemplo, manipular ao mesmo tempo 100 nomes de pessoas. Em vez de realizar a declaração de 100 variáveis, é possível a declaração de apenas uma: trata-se de uma variável definida como um vetor (*array*) de nomes.

Os *arrays* são utilizados para armazenar um conjunto que tenham o mesmo tipo de dado primitivo ou a mesma classe.

O *array* possibilita armazenar diversos valores em uma única variável, além do armazenamento de vários objetos.

Esses diversos itens são armazenados em forma de tabela de fácil manipulação, sendo diferenciados e referenciados por um índice numérico.

Os *arrays* estão presentes em praticamente todas as linguagens de programação e constituem um dos aspectos mais importantes e facilitadores no desenvolvimento de aplicações.

Em Java, os *arrays* são estruturas que permitem armazenar uma lista de itens relacionados.

6.2 *Arrays* unidimensionais

Os *arrays* unidimensionais são os que possuem apenas um índice para acessar seu conteúdo. Eles são declarados da seguinte maneira:

```
Tipo_de_dado nome_array[] = new tipo_dado[quantidade];
```

Tipo_do_dado: pode ser qualquer tipo de variável primitiva ou classe;

Nome_array: um nome qualquer válido, da mesma forma que os nomes das variáveis.

Por exemplo:

Int n[]=new int[100];

String mes[]=ney string[12];

Veja o Exemplo0601 que mostra a utilização de um *array* para armazenar um conjunto de argumentos do tipo inteiro, passado pelo usuário na linha de execução (Figuras 6.1 e 6.2).

Exemplo0601.java

```
1  class Exemplo0601
2  {
3      public static void main (String args[])
4      {
5          int i,total = 0;
6          int n[] = new int[10]; // declara um array n de 10 elementos int
7          if(args.length>0)  // executa o trecho se pelo menos um argumento foi
           fornecido
8          {
9              try
10             {
11                 for (i=0;i<args.length;i++)
12                 {   // armazena um inteiro no elemento do array
13                     n[i] = Integer.parseInt(args[i]);
14                     total = total + n[i]; // acumula a somatoria dos elementos
15                 }
16                 System.out.println("Os numeros digitados na ordem inversa foram:");
17                 for (i=args.length-1;i>=0;i--)
18                 {
19                     System.out.print(n[i] + " ");
20                 }
21                 System.out.println();
22                 System.out.println("Somatoria dos numeros = " + total);
23                 System.out.println("Media entre eles = " + (total / args.length));
24             }
25             catch(NumberFormatException erro)
26             {
27                 System.out.println("Erro, todos os argumentos devem ser numeros
                   inteiros");
28             }
29         }
30         else
31         {
32             System.out.println("Digite pelo menos um numero");
33         }
34     }
35 }
```

Figura 6.1: Exemplo0601
Fonte: Elaborada pelo autor

O Exemplo0601 recebe diversos números na linha de comando (no Máximo dez) e armazena-os em um *array* de números inteiros. Isso não é realizado diretamente, pois inicialmente os números são armazenados no *array args*(um valor de *strings*). O laço FOR se encarrega de converter os elementos do *array*, um a um. Armazenados no **array n**, os elementos são totalizados pela variável **total**.

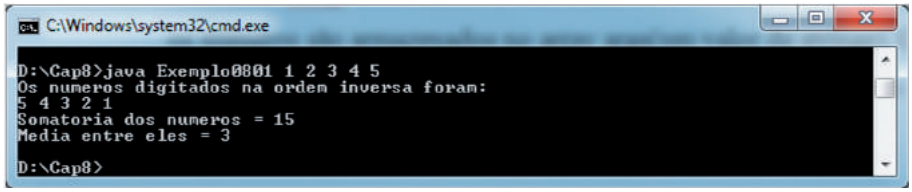


Figura 6.2: Execução do programa Exemplo0601
Fonte: Elaborada pelo autor

Os *arrays* podem ser criados e inicializados simultaneamente.

Em vez de usar o operador *new* para criar um objeto *array*, é preciso colocar os elementos do *array* entre chaves e separados por vírgula. Esses elementos dentro das chaves devem ser do mesmo tipo que a variável que contém o *array*. Os *arrays* criados dessa forma têm o mesmo tamanho do número de elementos colocados entre chaves. A sintaxe ficaria assim:

Tipo de dado nome_array[] = (valores separados por vírgula)

O Exemplo0602 a seguir demonstra como usar essa declaração e utiliza também o método **valueOf()** para manipular o conteúdo de um *array* de caracteres (Figuras 6.3 e 6.4).

Exemplo0602.java

```
1  class Exemplo0602
2  {
3    public static void main (String args[])
4    {
5      String nomes = "";
6      char caracterArray[] = {'a','b','c','d','e','f','g'};
7      System.out.println();
8      // criando um array de Strings
9      String stringArray[] = {"Aprendendo","a","linguagem","Java"};
10     for(int i=0;i<stringArray.length;i++)
11       nomes = nomes + stringArray[i] + " ";
12
13     JOptionPane.showMessageDialog(null, "array: " + String.valueOf(caracterArray) +
14       "\nQuant. de elementos: " + caracterArray.length +
15       "\n3 primeiros caracteres do array: " + String.valueOf(caracterArray,0,3) +
16       "\nMostrando o array: " + nomes +
17       "\nPrimeiro elemento: " + stringArray[0] +
18       "\núltimo elemento: " + stringArray[stringArray.length-1]
19     );
20   }
21 }
```

Figura 6.3: Exemplo0602
Fonte: Elaborada pelo autor

Existe uma diferença básica na atribuição de valores aos *arrays* de caracteres e de *strings*: nos *arrays* de caracteres são utilizados ‘’ apóstrofos para cada caractere declarado; já para os *arrays* de *strings* são utilizadas as “” aspas duplas.

O método **valueOf()** pode ser utilizado para apresentar todos os elementos de um *array* de caracteres ou um trecho dele.

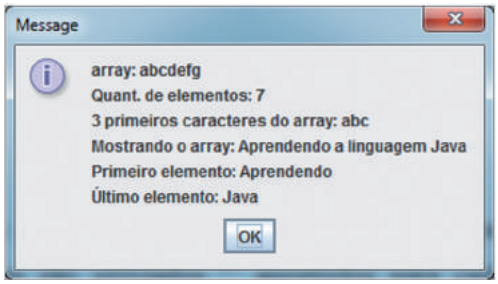


Figura 6.4: Execução do programa Exemplo0602
Fonte: Elaborada pelo autor

6.3 *Arrays* bidimensionais

Os *arrays* bidimensionais permitem a criação de vetores com mais de um índice. Essa característica possibilita que os valores sejam armazenados na forma de matriz de qualquer dimensão.

A linguagem Java não suporta *array* bidimensional no formato linha e coluna como em outra linguagem; entretanto, é possível criar *array* de *arrays*.

Esses *arrays* devem ser declarados da seguinte maneira:

Tipodedado nome_array[][] = new tipo_dado [índice][índice];

O Exemplo0603 a seguir demonstra o *array* bidimensional para coletar duas notas de três alunos. Uma vez armazenadas, o programa solicita ao usuário o número de um aluno para mostrar suas notas e a média do grupo de alunos (Figuras 6.5 e 6.6).

Exemplo0603.java

```
1  class Exemplo0603
2  {
3    public static void main (String args[])
4    {
5      float notas[][] = new float[3][2];
6      int aluno = 0,nota;
7      while(aluno<3)
8      {
9        nota = 0;
10       while(nota<2)
11       {
12         notas[aluno][nota] = Float.parseFloat(JOptionPane.showInputDialog(null, "Forneça a nota " + (
           nota+1) + " do aluno "+ (aluno+1)));
13         nota++;
14       }
15       aluno++;
16     }
17     aluno = Integer.parseInt(JOptionPane.showInputDialog(null, "Forneça o número do aluno a
       consultar: "));
18     JOptionPane.showMessageDialog(null, "CONSULTA DE NOTAS: " +
19       "\nAluno : " + aluno +
20       "\nNota1: " + notas[aluno - 1][0] +
21       "\nNota2: " + notas[aluno - 1][1] +
22       "\nMedia: " + ((notas[aluno - 1][0] + notas[aluno - 1][1]) / 2)
23     );
24     System.exit(0);
25   }
26 }
```

Figura 6.5: Exemplo0603
Fonte: Elaborada pelo autor

São coletadas e armazenadas duas notas de três alunos no *array* no formato de uma tabela. A nota fornecida pelo usuário é armazenada no vetor de notas.

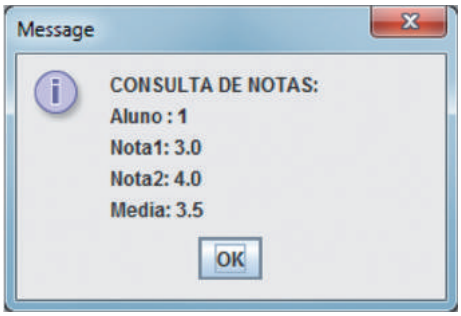


Figura 6.6: Execução do programa Exemplo0603
Fonte: Elaborada pelo autor

6.4 Passagem de *arrays* em métodos

É possível a criação de métodos que recebem valores, manipulam esses valores e retornam um resultado. Na passagem desses valores pode ser usado qualquer tipo de variável. Da mesma forma, é possível também criar métodos com passagem retorno de *arrays*. O funcionamento é basicamente o mesmo: quando o método é invocado, um *array* qualquer é passado, o qual é manipulado internamente pelo método e depois é retornado.

A sintaxe para um método que recebe e retorna um *array* é:

```
public static tipo_array[] nome_metodo (tipo_array nome_array[])
```

O Exemplo0604 a seguir mostra um método que recebe um *array* do tipo inteiro, organiza seus elementos e o retorna em ordem crescente (Figuras 6.7 e 6.8).

```java
import javax.swing.*;
class Exemplo0604
{
  public static void main (String args[])
  {
    int array[] = new int[5]; // cria um array com 5 elementos
    String saidaDigitada = "Array digitado:    ";
    for (int i=0;i<5;i++) // armazena os elementos no array
    {
      array[i] = Integer.parseInt(JOptionPane.showInputDialog(null, "Forneça o
      número " + (i + 1) + " entre 0 e 1000: "));
      saidaDigitada += array[i] + " - ";
    }
    array = ordenaArrayInt(array); // ordena o array
    String saidaOrdenada = "Array ordenado: ";
    for(int i=0;i<array.length;i++) // preenche a string de saida com o array
    ordenado
      saidaOrdenada += array[i] + " - ";
    JOptionPane.showMessageDialog(null,"\n" + saidaDigitada + "\n" + saidaOrdenada
    );
    System.exit(0);
  }

  public static int[] ordenaArrayInt(int arr[])
  {
    int x,y,aux;
    for(x=0;x<arr.length;x++)
      for (y=0;y<arr.length;y++)
      {
        if(arr[x]<arr[y])
        {
          aux=arr[y];
          arr[y]=arr[x];
          arr[x]=aux;
        }
      }
      return (arr);
  }
}
```

Figura 6.7: Exemplo0604
Fonte: Elaborada pelo autor

Figura 6.8: Execução do programa Exemplo0604
Fonte: Elaborada pelo autor

6.5 *Array* de objetos

Da mesma forma que variáveis primitivas, é possível criar um *array* para armazenamento de objetos.

Isso é muito importante na linguagem Java, pois permite realizar as mesmas operações com diversos objetos do mesmo tipo.

O Exemplo0605 demonstra a utilização de um *array* de objetos, aproveitando-se da funcionalidade de uma classe pessoa, conforme mostrado nas Figuras 6.9 e 6.10 a seguir.

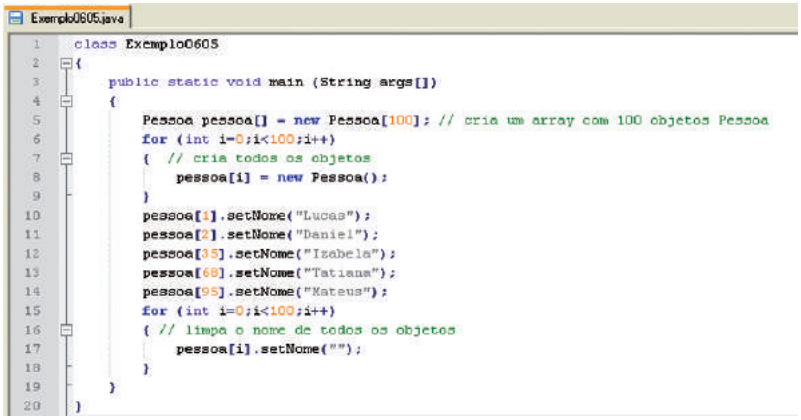


Figura 6.9: Exemplo0605
Fonte: Elaborada pelo autor

Observe que se os objetos fossem tratados de forma individual, isto é, cada um com nome diferente, seriam necessárias 100 linhas de código, uma para cada objeto.

Resumo

Nesta aula falamos sobre as principais formas de utilização de *arrays* em Java, desde as estruturas mais simples até estruturas complexas para tratar dados. Apresentamos as vantagens de usar *arrays* na manipulação de objetos e trabalhamos com conceitos que permitem ao programador desenvolver aplicações mais consistentes e com código reduzido.

Atividades de aprendizagem

1. Crie uma classe que leia dez valores inteiros quaisquer e imprima na tela os que são maiores que a média dos valores coletados.

2. Elabore uma classe que colete uma indefinida quantidade de números inteiros pela linha de execução e no final mostre o menor e o maior número fornecidos.

3. Faça uma classe que colete dez nomes de pessoas e os armazene em um *array*. No final verifique se uma determinada pessoa foi cadastrada no *array*, informando o usuário.

4. Uma escola precisa de um programa que controle a média das notas de cada classe e a média das notas de todos os alunos da escola. Considerando que essa escola possui três classes com cinco alunos em cada classe, gerando um total de 15 alunos, crie uma classe que receba as notas de cada aluno e no final mostre a média da classe e a média da escola em geral.

Aula 7 – Manipulando arquivos

Objetivos

Demonstrar a importância do armazenamento e recuperação de dados.

Enumerar os aspectos fundamentais para a leitura e gravação em arquivos.

Apresentar os passos necessários para armazenar arquivos no formato texto.

7.1 Definição

A grande maioria das aplicações necessita armazenar dados para manipulá-los posteriormente.

São poucas as aplicações que se limitam a armazenar dados na memória durante o processo de execução.

Os dados manipulados durante o processo de execução precisam ser recuperados a qualquer momento. Por esse motivo devem ser usados os **arquivos** de dados.

Para a manipulação de arquivos em Java, é necessária a utilização do pacote **java.io**.

Os dados podem ser armazenados e recuperados pelo pacote **java.io** por intermédio de um sistema de comunicação denominado controle de fluxo (*Stream*), permitindo a manipulação de diferentes formatos de arquivo, entre eles: txt, data, gif.

Arquivo
É um conjunto de dados armazenados em uma memória secundária não volátil que pode ser recuperado pelo programa a qualquer instante.

7.2 Leitura e gravação de um arquivo texto

Existem diversas maneiras de realizar a manipulação de arquivo texto. A forma apresentada utiliza a classe *BufferedReader* para a leitura do arquivo e classe *PrintWriter* para a gravação.

O Exemplo1101 a seguir demonstra o código necessário para a criação de um cadastro de pessoas usando a leitura e gravação em arquivo texto (Figuras 7.1 e 7.2). O nome do arquivo é código da pessoa mais a extensão txt e será armazenado na mesma pasta em que a classe estiver localizada.

Exemplo1101.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
class Exemplo1101 extends JFrame implements ActionListener
{
  JLabel label1, label2, label3;
  JButton btAbrir, btGravar,btLimpar;
  JTextField tfCodigo, tfNome, tfEmail;
  public static void main(String[] args)
  {
    JFrame janela = new Exemplo1101();
    janela.setUndecorated(true);
    janela.getRootPane().setWindowDecorationStyle(JRootPane.FRAME);
    janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    janela.setVisible(true);
  }
  Exemplo1101()
  {
    setTitle("Cadastro usando arquivo texto");
    setBounds (250,50,340,160);
    getContentPane().setBackground (new Color (150,150,150));
    label1 = new JLabel ("Código");
    label2 = new JLabel ("Nome");
    label3 = new JLabel ("Email");
    btAbrir = new JButton ("Abrir");
    btGravar = new JButton ("Gravar");
    btLimpar = new JButton ("Limpar");
    tfCodigo = new JTextField ();
    tfNome = new JTextField ();
    tfEmail = new JTextField ();
    btAbrir.addActionListener(this);
    btGravar.addActionListener(this);
    btLimpar.addActionListener(this);
    setLayout (null);
    label1.setBounds (10, 15, 40, 20);
    label2.setBounds (10, 40, 45, 20);
    label3.setBounds (10, 65, 45, 20);
    btAbrir.setBounds (10, 100, 75, 20);
    btGravar.setBounds (95, 100, 75, 20);
    btLimpar.setBounds (180, 100, 75, 20);
    tfCodigo.setBounds (60, 15, 55, 20);
    tfNome.setBounds (60, 40, 255, 20);
    tfEmail.setBounds (60, 65, 255, 20);
    getContentPane().add(label1);
    getContentPane().add(label2);
    getContentPane().add(label3);
    getContentPane().add(btAbrir);
    getContentPane().add(btGravar);
    getContentPane().add(btLimpar);
    getContentPane().add(tfCodigo);
    getContentPane().add(tfNome);
    getContentPane().add(tfEmail);
  }
```

```
56    public void actionPerformed(ActionEvent e)
57    {
58      if (e.getSource() == btLimpar)
59      {
60        tfCodigo.setText("");
61        tfNome.setText("");
62        tfEmail.setText("");
63      }
64
65      if (e.getSource() == btGravar)
66      {
67        if (tfCodigo.getText().equals(""))
68        {
69          JOptionPane.showMessageDialog(null, "O código não pode estar vazio!");
70          tfCodigo.requestFocus();
71        }
72        else if (tfNome.getText().equals(""))
73        {
74          JOptionPane.showMessageDialog(null, "O nome não pode estar vazio!");
75          tfNome.requestFocus();
76        }
77        else if (tfEmail.getText().equals(""))
78        {
79          JOptionPane.showMessageDialog(null, "O email não pode estar vazio!");
80          tfEmail.requestFocus();
81        }
82        else
83          try
84          {
85            PrintWriter out = new PrintWriter(tfCodigo.getText()+".txt");
86            out.println(tfCodigo.getText());
87            out.println(tfNome.getText());
88            out.println(tfEmail.getText());
89            out.close();
90            JOptionPane.showMessageDialog(null, "Arquivo gravado com sucesso!");
91          }
92          catch(IOException erro)
93          {
94            JOptionPane.showMessageDialog(null, "Erro ao gravar no arquivo");
95          }
96      }
97
98      if (e.getSource() == btAbrir)
99      {
100       try
101       {
102         String arq = JOptionPane.showInputDialog(null, "Forneça o código a abrir:");
103         BufferedReader br = new BufferedReader(new FileReader(arq+".txt"));
104         tfCodigo.setText(br.readLine());
105         tfNome.setText(br.readLine());
106         tfEmail.setText(br.readLine());
107         br.close();
108       }
109       catch(IOException erro)
110       {
111         JOptionPane.showMessageDialog(null, "Erro ao abrir o arquivo");
112       }
113     }
114   }
115 }
```

Figura 7.1: Exemplo1101
Fonte: Elaborada pelo autor

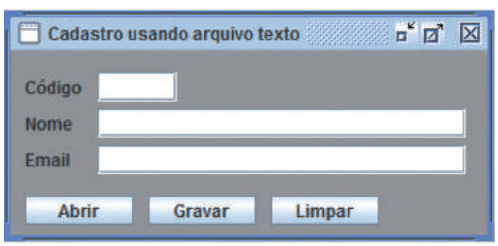


Figura 7.2: Execução do programa Exemplo1101
Fonte: Elaborada pelo autor

O pacote **java.io** é necessário para a manipulação de arquivos de fluxo.

O aplicativo do Exemplo1101 possui três botões: um para abrir texto, outro para gravar o arquivo texto e outro para limpar o conteúdo das caixas de texto, conforme indica a Figura 7.2. Ao pressionar o botão Gravar, o conteúdo das caixas de texto é armazenado num arquivo texto cujo nome é igual ao código da pessoa (mais a extensão .txt). Dessa forma, cada pessoa cadastrada é armazenada num arquivo texto diferente, isto é, a pessoa de código "10" é armazenada no arquivo "10.txt", a pessoa de código "15" é armazenada no arquivo "15.txt" e assim sucessivamente. Ao pressionar o botão Abrir, ocorre o processo inverso, isto é, o conteúdo armazenado no arquivo, cujo nome deve ser fornecido pelo usuário, é copiado para as caixas de texto. Ao pressionar o botão Limpar, o conteúdo das caixas de texto tfCodigo, tfNome e tfEmail é apagado.

A validação das caixas de texto é realizada ao pressionar o botão Gravar.

Todas as caixas de texto precisam ter algum conteúdo; caso contrário, uma mensagem é enviada ao usuário indicando a obrigatoriedade do preenchimento.

Observe que é utilizado o método **requestFocus()**, responsável por fornecer o foco (do cursor) ao objeto correspondente.

O Exemplo1101 contém a classe **PrintWriter** usada para criar o arquivo **texto**, cujo nome é o conteúdo da caixa de texto tfcodigo concatenado com a extensão ".txt". O arquivo a ser criado é controlado pelo objeto *out*. O objeto *out* realiza a gravação dos valores do código, nome e *e-mail*, cada um em uma linha diferente do arquivo texto por meio do método *println*, que faz com que cada dado gravado ocupe uma linha diferente do arquivo. Imagine que você está criando um arquivo texto por meio do bloco de notas e cada linha receberá um dado diferente.

O método **close()** é responsável por fechar o arquivo. Os dados são efetivamente transferidos para o arquivo texto quando o objeto *out* é fechado.

A exceção **IOException** é gerada quando, por um motivo qualquer, não for possível realizar a gravação do arquivo. Em resumo, para o armazenamento de dados em um arquivo texto, é necessário:

a) criar um arquivo de fluxo para a saída de dados por meio da classe **PrintWriter**;

b) gravar os dados no arquivo por meio do método **println()**;

c) fechar o arquivo gravado por meio do método **close()**.

A abertura e leitura de um arquivo texto são executadas quando o usuário pressiona o botão Abrir. O objeto *br* é criado como um objeto da classe **BufferedReader**. Observe que o objeto tenta abrir o arquivo cujo nome foi solicitado ao usuário. Caso haja sucesso na abertura do arquivo, o cursor fica posicionado na primeira linha do arquivo. Isso também ocorre quando um arquivo de texto é aberto por meio do bloco de notas.

Experimente abrir o arquivo.txt com o bloco de notas e verifique que o cursor fica posicionado no primeiro caractere da primeira linha.

A leitura do conteúdo do arquivo texto, linha a linha, é realizada por meio do método **readLine()**, que o armazena nas caixas de texto correspondentes o código, nome e *e-mail*. Cada vez que o método **readLine()** é executado, uma linha do arquivo é lida e o cursor é posicionado automaticamente na próxima linha.

O Exemplo1102 a seguir apresenta uma aplicação que funciona como um editor de textos bem básico. Ele realiza a leitura ou a gravação de um arquivo texto qualquer, escolhido pelo usuário por meio da caixa de diálogo da classe **FileDialog**.

Quando o usuário pressiona o botão Gravar ou o botão Abrir, aparece uma caixa de diálogo semelhante à utilizada pelo Windows, por exemplo, no bloco de notas. Com isso, torna-se possível realizar a escolha do arquivo que será lido ou gravado pela aplicação. O Exemplo1102 mostra também outras duas classes (**FileWriter** e **FileReader**) que podem ser usadas para a manipulação de arquivos texto. O código do exemplo é apresentado na Figura 7.3 e a Figura 7.4 exibe a execução do programa.

```
54    public void actionPerformed(ActionEvent e)
55    {
56      String nome_do_arquivo;
57      if (e.getSource() == btLimpar)
58      {
59        textArea1.setText("");
60        tfTexto.setText("");
61      }
62      if (e.getSource() == btGravar)
63      {
64        try
65        {
66          fdSalvar.setVisible(true);
67          if (fdSalvar.getFile()==null) return;
68          nome_do_arquivo = fdSalvar.getDirectory()+fdSalvar.getFile();
69          FileWriter out = new FileWriter(nome_do_arquivo);
70          out.write(textArea1.getText());
71          out.close();
72          tfTexto.setText("Arquivo gravado com sucesso !");
73        }
74        catch(IOException erro)
75        {
76          tfTexto.setText("Erro ao gravar no arquivo !");
77        }
78      }
79      if (e.getSource() == btAbrir)
80      {
81        try
82        {
83          fdAbrir.setVisible(true);
84          if (fdAbrir.getFile()==null) return;
85          nome_do_arquivo = fdAbrir.getDirectory()+fdAbrir.getFile();
86          FileReader in = new FileReader(nome_do_arquivo);
87          String s = "";
88          int i = in.read();
89          while (i!=-1)
90          {
91            s = s +(char)i;
92            i = in.read();
93          }
94          textArea1.setText(s);
95          in.close();
96          tfTexto.setText("Arquivo aberto com sucesso !");
97        }
98        catch(IOException erro)
99        {
100         tfTexto.setText("Erro ao abrir o arquivo !");
101       }
102     }
103   }
104 }
```

Figura 7.3: Exemplo1102

Fonte: Elaborada pelo autor

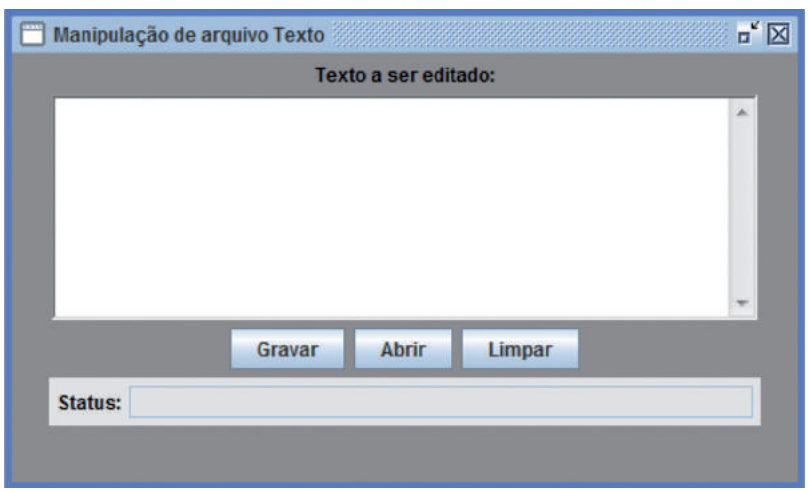


Figura 7.4: Execução do programa Exemplo1102

Fonte: Elaborada pelo autor

No Exemplo1102 são declarados dois objetos (fdAbrir e fdSalvar) como da classe **FileDialog**, usada na criação de caixas de diálogo para manipulação de arquivos.

O comando fdAbrir = new FileDialog (this, "Abrir arquivo", FileDialog.LOAD); inicializa o objeto fdAbrir como uma caixa de diálogo com o título "Abrir arquivo" para a leitura de arquivos (FileDialog.LOAD).

O comando fdSalvar = new FileDialog (this, "salvar arquivo" , FileDialog.SAVE); inicializa o objeto fdSalvar como uma caixa de diálogo com o título "Salvar arquivo" para a gravação de arquivos (FileDialog.SAVE). A abertura da caixa de diálogo de gravação pelo método denominado **setVisible(*true*)**.

O comando if (fdSalvar.getFile()==null) return; no momento de realizar a gravação do arquivo, é aberta uma caixa de diálogo. Para descobrir o arquivo escolhido pelo usuário, é utilizado o método **getFile**. Caso o usuário não tenha escolhido nenhum arquivo, ou se pressionou o botão Cancelar da caixa de diálogo, o método **getFile** retorna *null*, o que provoca o encerramento do método **actionPerformed** por meio de *return* e, consequentemente, a não gravação do arquivo. Caso contrário, o caminho e o nome do arquivo são armazenados na variável nome_do_arquivo pelos métodos **getDirectory** e **getFile**, de acordo com:

```
nome_do_arquivo = fdsalvar.getDirectory() + fdsalvar.getFile();
```

É realizada a inicialização de *out* como um objeto da classe **FileWriter**, apontando para a variável nome_do_arquivo que contém o arquivo escolhido ou digitado pelo usuário.

Todo o conteúdo do TextArea é armazenado no arquivo pelo método **write**.

A abertura da caixa de diálogo de leitura pelo método **setVisible(true)**.

É inicializado *in* como um objeto da classe **FileReader** apontando para a variável nome_do_arquivo que contém o arquivo escolhido ou digitado pelo usuário.

No processo de leitura é utilizada a classe **FileReader**, por meio do método **read**, cada caractere é lido como um inteiro.

Quando o final do arquivo for encontrado, é retornado o valor -1.

Também é verificado se o ponteiro já se encontra no final do arquivo, ou seja, enquanto **i** for diferente de -1, o arquivo é lido caractere a caractere, cada caractere é lido (tipo inteiro) e convertido no tipo *char*, sendo acumulado na **String s**. Dessa forma, realiza-se o processo de leitura do primeiro ao último caractere do arquivo texto escolhido.

Resumo

Nesta aula falamos sobre as principais formas de utilização de arquivos em Java, desde as estruturas mais simples até estruturas complexas para tratar dados e salvar em arquivos. Enumeramos os aspectos fundamentais para a leitura e gravação em arquivos. Demonstramos a importância do armazenamento e da recuperação de dados. Apresentamos os passos necessários para armazenar arquivos no formato texto.

Atividades de aprendizagem

1. Crie uma classe para armazenar que tenha as seguintes opções na manipulação de arquivo: Abrir, gravar e limpar. Os dados utilizados devem ser: matrícula, nome_aluno, endereço, telefone e CEP.

2. Crie uma classe para simular um editor de textos básico e tente implementar algumas opções extras diferentes dos exemplos aqui apresentados.

Aula 8 – Estruturas de dados em Java: listas

Objetivos

Demonstrar a importância do armazenamento e recuperação em estruturas de dados.

Enumerar os aspectos fundamentais para a utilização de listas lineares.

Apresentar os passos necessários para implementação de listas.

8.1 Definição de listas

Uma das formas mais fáceis de interligar elementos de um conjunto é por meio de uma lista.

A lista é um tipo de estrutura de dados bastante flexível, porque permite o crescimento ou redução de seu tamanho durante a execução do programa.

Os itens da lista podem ser acessados, retirados ou inseridos. Podemos também juntar duas listas para formar uma única, assim como uma lista pode ser partida em duas ou mais listas.

As listas são utilizadas para aplicações nas quais não é possível prever a utilização de memória, permitindo a manipulação de quantidade imprevisível de dados.

As listas são úteis em aplicações tais como manipulação simbólica, gerência de memória, simulação e compiladores.

Uma lista linear é uma sequência de zero ou mais itens:

x1, x2, · , xn, na qual xi é de um determinado tipo e n representa o tamanho da lista linear.

Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão:

- assumindo $n \geq 1$, x_1 é o primeiro item da lista e x_n é o último item da lista
- x_i precede x_{i+1} para $i = 1, 2, \cdots, n - 1$
- x_i sucede x_{i-1} para $i = 2, 3, \cdots, n$
- o elemento x_i é dito estar na i-ésima posição da lista.

Para criar um tipo abstrato de dados Lista em Java, é necessário definir um conjunto de operações sobre os objetos do tipo lista. O conjunto de operações pode ser definido e depende de cada aplicação.

Um conjunto de operações necessárias a uma maioria de aplicações é:

a) Criar uma lista linear vazia.

b) Inserir um novo item imediatamente após o i-ésimo item.

c) Retirar o i-ésimo item.

d) Localizar o i-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.

e) Combinar duas ou mais listas lineares em uma lista única.

f) Partir uma lista linear em duas ou mais listas.

g) Fazer uma cópia da lista linear.

h) Ordenar os itens da lista em ordem ascendente ou descendente, de acordo com alguns de seus componentes.

i) Pesquisar a ocorrência de um item com um valor particular em algum componente.

8.2 Implementação de listas por meio de arranjos

Em um tipo estruturado arranjo, os itens da lista são armazenados em posições contíguas de memória.

Os itens da lista são armazenados em posições contíguas de memória.

Algumas características de arranjos:

- A lista pode ser percorrida em qualquer direção.
- A inserção de um novo item pode ser realizada após o último item com custo constante.
- A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção.
- Retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.

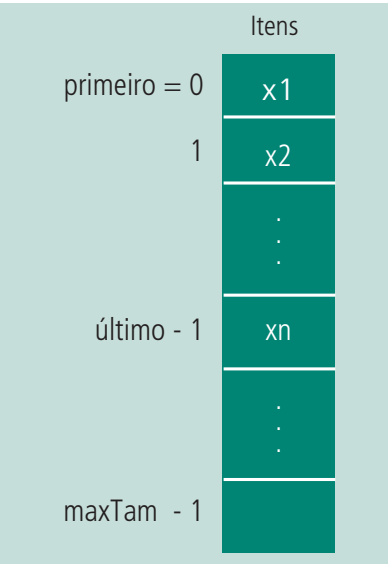


Figura 8.1: Lista utilizando arranjo
Fonte: Elaborada pelo autor

A estrutura lista utilizando arranjo é uma possível implementação para as nove operações definidas anteriormente para o tipo abstrato de dados lista. No programa mostrado na Figura 8.2 utilizamos a estrutura lista em que o campo item é o principal componente da classe **Lista**.

listaEncadeada.java

```
1  package cap3.arranjo;
2  public class Lista {
3      private Object item[ ];
4      private int primeiro , ultimo , pos;
5      //Operações
6      public Lista ( int maxTam) { // Cria uma Lista vazia
7          this.item = new Object [maxTam] ; this.pos = -1;
8          this.primeiro = 0; this.ultimo = this.primeiro;
9      }
10     public Object pesquisa (Object chave) {
11         if ( this.vazia ( ) | | chave == null ) return null;
12         for ( int p = 0; p < this.ultimo; p++)
13         if ( this.item[p].equals (chave) ) return this.item[p];
14         return null;
15     }
16     public void insere (Object x) throws Exception {
17         if ( this.ultimo >= this.item.length)
18         throw new Exception ( "Erro : A l ista esta cheia" );
19         else {
20             this.item[ this.ultimo ] = x;
21             this.ultimo = this.ultimo + 1;
22         }
23     }
24     public Object retira (Object chave) throws Exception {
25         if ( this.vazia ( ) | | chave == null )
26         throw new Exception ( "Erro : A l ista esta vazia" );
27         int p = 0;
28         while(p < this.ultimo && !this.item[p].equals(chave) ) p++;
29         if (p >= this.ultimo ) return null ; // Chave não encontrada
30         Object item = this.item[p] ;
31         this.ultimo = this.ultimo - 1;
32         for ( int aux = p; aux < this.ultimo; aux++)
33         this.item[aux] = this.item[aux + 1];
34         return item;
35     }
36     public Object retiraPrimeiro ( ) throws Exception {
37         if ( this.vazia ( ) ) throw new Exception
38         ( "Erro : A l ista esta vazia" ) ;
39         Object item = this.item[0];
40         this.ultimo = this.ultimo - 1;
41         for ( int aux = 0; aux < this.ultimo; aux++)
42         this.item[aux] = this.item[aux + 1];
43         return item;
44     }
45     public Object primeiro ( ) {
46         this.pos = -1; return this.proximo ( );
47     }
48     public Object proximo ( ) {
49         this.pos++;
50         if ( this.pos >= this.ultimo ) return null;
51         else return this.item[ this.pos] ;
52     }
53     public boolean vazia ( ) {
54         return ( this.primeiro == this.ultimo );
55     }
56     public void imprime ( ) {
57         for ( int aux = this.primeiro; aux < this.ultimo; aux++)
58         System.out.println ( this.item[aux].toString ( ) );
59     }
60 }
```

Figura 8.2: Lista com arranjo
Fonte: Elaborada pelo autor

Os itens são armazenados em um arranjo de tamanho suficiente para armazenar a lista.

O i-ésimo item da lista está armazenado na i-ésima posição do arranjo, 1≤ i <Último.

A constante **MaxTam** define o tamanho máximo permitido para a lista.

8.2.1 Lista utilizando arranjo: vantagem e desvantagens

Vantagem:

- Economia de memória (os apontadores são implícitos nessa estrutura).

Desvantagens:

- Custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens, no pior caso.
- Em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos deve ter a realocação de memória.

8.3 Implementação de listas por meio de estruturas autorreferenciadas

Em uma estrutura autorreferenciada, cada item da lista contém a informação que é necessária para alcançar o próximo item.

Esse tipo de implementação permite utilizar posições não contíguas de memória, sendo possível inserir e retirar elementos sem haver a necessidade de deslocar os itens seguintes da lista.

Há uma célula cabeça para simplificar as operações sobre a lista (Figura 8.3).

Figura 8.3: Lista utilizando estruturas autorreferenciadas
Fonte: Elaborada pelo autor

A estrutura da lista utilizando estruturas autorreferencidas e uma possível implementação para as nove operações definidas anteriormente para o tipo lista são mostradas no exemplo mostrado na Figura 8.4 a seguir.

listaAutoReferencia.java

```
1  package cap3.autoreferencia;
2  public class Lista {
3      private static class Celula { Object item; Celula prox ; }
4      private Celula primeiro , ultimo , pos;
5      // Operações
6      public Lista ( ) { // Cria uma Lista vazia
7          this.primeiro = new Celula ( ) ; this.pos = this.primeiro;
8          this.ultimo = this.primeiro ; this.primeiro.prox = null;
9      }
10     public Object pesquisa (Object chave) {
11         if ( this.vazia ( ) | | chave == null ) return null;
12         Celula aux = this.primeiro;
13         while (aux.prox != null ) {
14             if (aux.prox.item.equals (chave) ) return aux.prox.item;
15             aux = aux.prox;
16         } return null;
17     }
18     public void insere (Object x ) {
19         this.ultimo.prox = new Celula ( ) ;
20         this.ultimo = this.ultimo.prox;
21         this.ultimo.item = x ; this.ultimo.prox = null;
22     }
23     public Object retira (Object chave) throws Exception {
24         if ( this.vazia ( ) | | ( chave == null ) )
25         throw new Exception
26         ( "Erro : Lista vazia ou chave invalida" );
27         Celula aux = this.primeiro;
28         while (aux.prox!=null && !aux.prox.item.equals(chave) )
29         aux=aux.prox;
30         if (aux.prox == null ) return null ; // não encontrada
31         Celula q = aux.prox;
32         Object item = q.item; aux.prox = q.prox;
33         if (aux.prox == null ) this.ultimo = aux; return item;
34     }
35     public Object retiraPrimeiro ( ) throws Exception {
36         if ( this.vazia ( ) ) throw new Exception
37         ( "Erro : Lista vazia" ) ;
38         Celula aux = this.primeiro ; Celula q = aux.prox;
39         Object item = q.item; aux.prox = q.prox;
40         if (aux.prox == null ) this.ultimo = aux; return item;
41     }
42     public Object primeiro ( ) {
43         this.pos = primeiro ; return proximo ( );
44     }
45     public Object proximo ( ) {
46         this.pos = this.pos.prox;
47         if ( this.pos == null ) return null;
48         else return this.pos.item;
49     }
50     public boolean vazia ( ) {
51         return ( this.primeiro == this.ultimo );
52     }
53     public void imprime ( ) {
54         Celula aux = this.primeiro.prox;
55         while (aux != null ) {
56         System.out.Println (aux.item.toString ( ) );
57         aux = aux.prox ; }
58     }
59 }
```

Figura 8.4: Exemplo de Lista com estrutura autoreferenciadas
Fonte: Elaborada pelo autor

8.3.1 Características principais das listas autoreferenciadas

A lista é constituída de células:

- Cada célula contém um item da lista e uma referência para a célula seguinte.

– A classe Lista contém uma referência para a célula cabeça, uma referência para a última célula da lista e uma referência para armazenar a posição corrente na lista.

8.3.2 Lista utilizando estruturas autorreferenciadas: vantagens e desvantagem

Vantagens:

– Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem).
– É boa para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido a priori).

Desvantagem:

– Utilização de memória extra para armazenar as referências.

Resumo

Nesta aula falamos sobre as principais formas de utilização de estruturas de dados em Java. Programamos a utilização da estrutura de dados lista na forma de arranjos e estrutura autorreferenciadas.

Atividades de aprendizagem

1. Refazer a estrutura do exemplo na Figura 8.2 para sempre permitir a inserção de novos elementos na lista. Para isso devemos alterar a operação Insere e toda vez que a inserção de um novo item esgotar a memória disponível pelo arranjo item, uma nova área de memória com capacidade maior deve ser alocada e o conteúdo do arranjo item deve ser copiado para ela.

2. Considerada a implementação de listas lineares utilizando estruturas autorreferenciadas e com a célula cabeça, escreva um método em Java para a classe lista que retorna *true* caso o valor ch estiver na lista e retorna *false* se o valor ch não estiver na lista.

Aula 9 – Estruturas de dados em Java: pilha

Objetivos

Demonstrar a importância do armazenamento e recuperação utilizando estruturas de dados.

Enumerar os aspectos fundamentais para a utilização de pilhas.

Apresentar os passos necessários para implementação de pilhas.

9.1 Definição de pilha

Existem aplicações para listas lineares nas quais inserções, retiradas e acesso a itens ocorrem sempre em um dos extremos da lista.

Pilha é uma lista linear em que todas as inserções, retiradas e acessos são feitos em apenas um extremo da lista.

Os itens em uma pilha são colocados um sobre o outro, com o item inserido mais recentemente no topo e o item inserido menos recentemente no fundo.

O modelo padrão da pilha é o de um monte de pratos em uma prateleira, sendo conveniente retirar pratos ou adicionar novos pratos na parte superior.

9.2 Propriedades e aplicações das pilhas

Propriedades:

- O último item inserido é o primeiro item que pode ser retirado da lista. São as chamadas listas LIFO ("last-in, first-out").
- Existe uma ordem linear para pilhas, do "mais recente para o menos recente".
- É ideal para processamento de estruturas aninhadas de profundidade imprevisível.
- Uma pilha contém uma sequência de obrigações adiadas. A ordem de remoção garante que as estruturas mais internas serão processadas antes das mais externas.

Aplicações em estruturas aninhadas:

- Quando é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente.
- O controle de sequências de chamadas de subprogramas.
- A sintaxe de expressões aritméticas.
- As pilhas ocorrem em estruturas de natureza recursiva (como árvores). Elas são utilizadas para implementar a recursividade.

9.3 Conjunto de operações

a) Criar uma pilha vazia.

b) Verificar se a lista está vazia. Retorna *true* se a pilha está vazia; caso contrário, retorna *false*.

c) Empilhar o item x no topo da pilha.

d) Desempilhar o item x no topo da pilha, retirando-o da pilha.

e) Verificar o tamanho atual da pilha.

Veja o exemplo mostrado na Figura 9.1 a seguir.

pilha.java

```
1  package cap3.arranjo;
2  public class Pilha {
3      private Object item[ ] ;
4      private int topo;
5      // Operações
6      public Pilha ( int maxTam) { // Cria uma Pilha vazia
7          this.item = new Object [maxTam] ; this.topo = 0;
8      }
9      public void empilha (Object x) throws Exception {
10         if ( this.topo == this.item.length)
11         throw new Exception ( "Erro : A pilha esta cheia" );
12         else this.item[ this.topo++] = x;
13     }
14     public Object desempilha ( ) throws Exception {
15         if ( this.vazia( ) )
16         throw new Exception ( "Erro : A pilha esta vazia" );
17         return this.item[--this.topo ]/
18     }
19     public boolean vazia ( ) {
20         return ( this.topo == 0);
21     }
22     public int tamanho ( ) {
23         return this.topo;
24     }
25 }
```

Figura 9.1: Exemplo de pilha com arranjo
Fonte: Elaborada pelo autor

9.4 Implementação de pilhas por meio de arranjo

Em uma implementação por meio de arranjos os itens da pilha são armazenados em posições contiguas de memória. Por causa das características da pilha, as operações de inserção e de retirada de itens devem ser implementadas de forma diferente da implementação feita com listas.

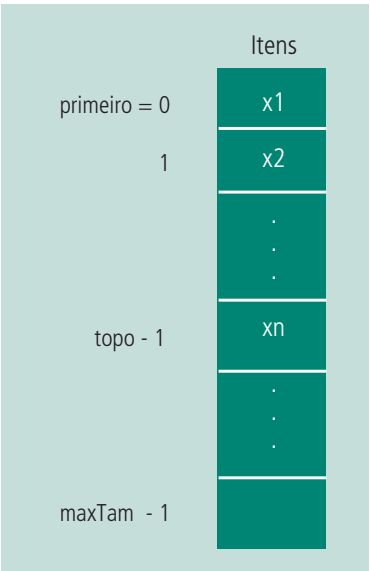


Figura 9.2: Pilha utilizando arranjo
Fonte: Elaborada pelo autor

Como as retiradas e inserções ocorrem no topo da pilha, um cursor chamado topo é utilizado para controlar a posição do item no topo da pilha.

Na estrutura da pilha usando arranjo, o campo item é o principal componente da classe pilha mostrado no exemplo abaixo:

9.5 Implementação de pilhas por meio de estruturas autorreferenciadas

Ao contrário da implementação de listas lineares por meio de estruturas autorreferenciadas, não há necessidade de manter uma célula cabeça no topo da pilha.

Para desempilhar um item, basta desligar a célula que contém xn, e a célula que contém xn–1 passa a ser a célula de topo.

Para empilhar um novo item, basta fazer a operação contrária, criando uma nova célula para receber o novo item (Figura 9.3).

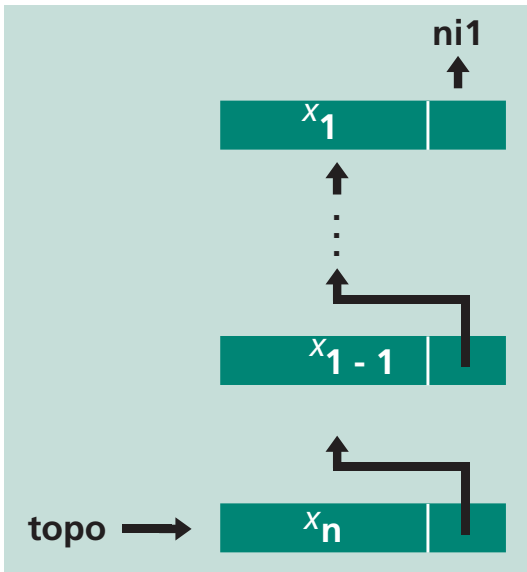


Figura 9.3: Pilha utilizando autoreferência
Fonte: Elaborada pelo autor

Algumas características da estrutura e operações sobre pilhas utilizando estruturas autorreferenciadas:

- O campo **tam** evita a contagem do número de itens no método tamanho.
- Cada célula de uma pilha contém um item da pilha e uma referência para outra célula.
- A classe **Pilha** contém uma referência para o topo da pilha.

pilhaAutoreferencia.java

```
1  package cap3.autorreferencia; public class Pilha {
2      private static class Celula {
3          Object item;
4          Celula prox;
5      }
6      private Celula topo;
7      private int tam;
8      // Operações
9      public Pilha ( ) { // Cria uma Pilha vazia
10         this.topo = null ; this.tam = 0;
11     }
12     public void empilha (Object x ) {
13         Celula aux = this.topo;
14         this.topo = new Celula ( ) ;
15         this.topo.item = x;
16         this.topo.prox = aux;
17         this.tam++;
18     }
19     public Object desempilha ( ) throws Exception {
20         if ( this.vazia ( ) )
21         throw new Exception ( "Erro : A pilha esta vazia" ) ;
22         Object item = this.topo.item;
23         this.topo = this.topo.prox;
24         this.tam--;
25         return item;
26     }
27     public boolean vazia ( ) {
28         return ( this.topo == null ) ;
29     }
30     public int tamanho ( ) {
31         eturn this.tam;
32     }
33 }
```

Figura 9.4: Exemplo de pilha com estrutura autoreferenciada
Fonte: Elaborada pelo autor

Resumo

Nesta aula falamos sobre as principais formas de utilização de estruturas de dados pilha em Java. Programamos a utilização da estrutura de dados pilha na forma de arranjos e estruturas autorreferenciadas.

Atividades de aprendizagem

1. Duas pilhas podem coexistir em um mesmo vetor, uma crescendo em um sentido, e a outra, no outro?

2. Duas pilhas podem ser alocadas no mesmo vetor com o mesmo grau de eficiência? Por quê?

3. É possível resolver o problema da representação por alocação sequencial para mais de duas pilhas?

Referências

DEITEL, H. M.; DEITEL, P. J. **Java:** como programar. Tradução e revisão técnica de Carlos Arthur Lang Lisbôa. 4. ed. Porto Alegre: Editora Bookman, 2003.

ARNOLD, Ken; GOSLING, James; HOLMES, David. **A linguagem de programação Java**. 4. ed. Porto Alegre: Editora Bookman, 2007.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estrutura de dados e algoritmos em Java**. 4. ed. Porto Alegre: Editora Bookman, 2007.

BONAN, Adilson Rodrigues. **Java. fundamentos, práticas & certificações**. São Paulo: Editora Alta Books, 2008.

FURGERI, Sérgio. **Java 6:** ensino didático: desenvolvendo e implementando aplicações. 6. ed. São Paulo: Editora Érica, 2008.

MOREIRA NETO, Eziel. E**ntendendo e dominando o Java**. 3. ed. São Paulo: Editora Digerati Books, 2009.

MOREIRA NETO, Eziel. **Entendendo e dominando o Java para internet**. 2. ed. São Paulo: Editora Digerati Books, 2009.

LIGUORI, Robert; LIGUORI, Patricia. **Java:** guia de bolso. São Paulo: Editora Alta Books, 2008.

Currículo do professor-autor

Antonio Luiz Santana é graduado em Engenharia Mecânica com ênfase em produção, com especialização em Redes de Computadores e Análise de Sistemas. Professor de algumas instituições de ensino superior e curso técnico desde 1991, lecionando disciplinas para o Curso Técnico em Informática e Superior de Tecnologia em Redes de Computadores e Análise de sistemas. Em EaD, é o responsável pelas disciplinas Técnicas de Programação, Sistemas de Informação I e Sistemas de Informação II, Serviços de Redes. Atua na área de Informática e redes desde 1992, quando se formou Técnico em Processamento de Dados pelo SENAC. Participou da elaboração do projeto do Curso Superior de Sistemas de Informação e dos Cursos Técnicos em Informática (modalidade presencial e EaD). Também é coordenador de TI e Redes de computadores do Campus Vitória do IFES.

ISBN 978-85-62934-01-8