

# Conceitos Avançados sobre Algoritmos e Programação com a Linguagem Java

Bruno Ferreira



Formação Inicial e  
Continuada

**+ IFMG**

*Campus Formiga*



Bruno Ferreira

**Conceitos Avançados sobre Algoritmos e Programação com a  
Linguagem Java**  
1ª Edição

Belo Horizonte

Instituto Federal de Minas Gerais

2022

© 2022 by Instituto Federal de Minas Gerais

Todos os direitos autorais reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico. Incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização por escrito do Instituto Federal de Minas Gerais.

Pró-reitor de Extensão	Carlos Bernardes Rosa Júnior
Diretor de Programas de Extensão	Nilton Vieira Junior
Coordenação do curso	Bruno Ferreira
Arte gráfica	Ângela Bacon
Diagramação	Eduardo dos Santos Oliveira

#### FICHA CATALOGRÁFICA

##### Dados Internacionais de Catalogação na Publicação (CIP)

---

F383c Ferreira, Bruno.

Conceitos avançados sobre algoritmos e programação com a linguagem Java / Bruno Ferreira – Belo Horizonte: Instituto Federal de Minas Gerais, 2022.

84 p. : il. color.

E-book, no formato PDF.

Material didático para Formação Inicial e Continuada.

ISBN 978-65-5876-041-2

1. Algoritmo. 2. Estrutura de dados. 3. Arquivos. I. Ferreira, Bruno. II. Título.

CDD 005.1

---

**Catalogação: Simoni Júlia da Silveira - CRB-6/2396**

Índice para catálogo sistemático:

Linguagens de programação: Processamento de dados:005.13

2022

Direitos exclusivos cedidos ao  
Instituto Federal de Minas Gerais  
Avenida Mário Werneck, 2590,  
CEP: 30575-180, Buritis, Belo Horizonte – MG,  
Telefone: (31) 2513-5157

## Sobre o material

Este curso é autoexplicativo e não possui tutoria. O material didático, incluindo suas videoaulas, foi projetado para que você consiga evoluir de forma autônoma e suficiente.

Caso opte por imprimir este *e-book*, você não perderá a possibilidade de acessar os materiais multimídia e complementares. Os *links* podem ser acessados usando o seu celular, por meio do glossário de Códigos QR disponível no fim deste livro.

Embora o material passe por revisão, somos gratos em receber suas sugestões para possíveis correções (erros ortográficos, conceituais, *links* inativos etc.). A sua participação é muito importante para a nossa constante melhoria. Acesse, a qualquer momento, o Formulário “Sugestões para Correção do Material Didático” clicando nesse [link](#) ou acessando o QR Code a seguir:



Formulário de  
Sugestões

Para saber mais sobre a Plataforma +IFMG acesse

<http://mais.ifmg.edu.br>



## Palavra(s) do(s) autor(es)

Seja bem-vindo ao curso de Conceitos Avançados sobre Algoritmos e Programação com a Linguagem Java.

A área de desenvolvimento de sistemas é extremamente atrativa, seja pelo grande número de vagas de empregos disponíveis ou pelos ótimos salários ofertados por esse campo de trabalho. No entanto, os profissionais dessa área devem dominar, além dos elementos básicos, o armazenamento de dados em vetores e matrizes. O programador deve saber como organizar o código de forma profissional através de procedimento e funções e, além disso, dominar o armazenamento e recuperação de informações em arquivos textos.

Assim, esse curso começa apresentando os conceitos básicos sobre vetores e matrizes.

Em seguida, vamos conhecer o conceito de procedimentos e funções. Vamos aprender como criar e usar esses elementos tão importantes para organização e reaproveitamento de código fonte.

Finalmente, vamos nos concentrar nas instruções que possibilitam armazenar dados de forma não volátil e por fim, como manipular essas informações.

Bons estudos!

**Bruno Ferreira.**





## Apresentação do curso

Este curso está dividido em quatro semanas, cujos objetivos de cada uma são apresentados, sucintamente, a seguir.

<b>SEMANA 1</b>	Definição e manipulação de Vetores.
<b>SEMANA 2</b>	Definição e manipulação de Matrizes
<b>SEMANA 3</b>	Entender, criar e executar Métodos (funções e procedimentos).
<b>SEMANA 4</b>	Entender, criar e manipular arquivos dos tipos binário e texto.

**Carga horária:** 40 horas.

**Estudo proposto:** 2h por dia em cinco dias por semana (10 horas semanais).



## Apresentação dos Ícones

Os ícones são elementos gráficos para facilitar os estudos, fique atento quando eles aparecem no texto. Veja aqui o seu significado:



**Atenção:** indica pontos de maior importância no texto.



**Dica do professor:** novas informações ou curiosidades relacionadas ao tema em estudo.



**Atividade:** sugestão de tarefas e atividades para o desenvolvimento da aprendizagem.



**Mídia digital:** sugestão de recursos audiovisuais para enriquecer a aprendizagem.



## Sumário

Semana 1 – Vetores.....	15
1.1. Considerações iniciais.....	15
1.2. Declaração/Instanciação de vetores.....	17
1.3. Atribuição de valores e leitura de dados.....	19
1.4. Percorrendo o vetor.....	21
1.5. Descobrindo o tamanho do vetor .....	23
1.6. Exemplos .....	25
Semana 2 – Matrizes .....	29
2.1 Considerações iniciais.....	29
2.2 Declaração/Instanciação de matrizes .....	31
2.3 Atribuição de valores e leitura de dados.....	34
2.4 Percorrendo a matriz.....	36
2.5 Descobrindo o tamanho da matriz.....	38
2.6 Exemplos .....	40
Semana 3 – Métodos (funções e procedimentos) .....	45
3.1 Considerações iniciais.....	45
3.2 Declaração .....	46
3.3 Ativação e fluxo de execução .....	50
3.4 Variáveis locais .....	51
3.5 Parâmetros de entrada.....	52
3.6 Exemplos .....	55
Semana 4 – Manipulação de arquivos.....	59
4.1 Conceitos iniciais .....	59
4.2 A classe File.....	61
4.3 Gravando dados em arquivos binários .....	65
4.4 Lendo dados de arquivos binários.....	66
4.5 Lendo dados de arquivos texto .....	67
4.6 Gravando dados de arquivos texto.....	68
4.7 Simplificando o acesso a arquivos do tipo texto .....	69

Referências .....	73
Currículo do autor.....	75
Glossário de códigos QR ( <i>Quick Response</i> ) .....	77



## Objetivos

Nesta primeira semana, você irá conhecer os conceitos básicos sobre vetores e também como manipulá-los para gerenciar múltiplas informações do mesmo tipo de dado.



**Mídia digital:** Antes de iniciar os estudos, vá até a sala virtual e assista ao vídeo “Apresentação do curso”.

## 1.1. Considerações iniciais

Como primeiro assunto no curso, nós vamos aprender sobre vetores. Acredito que se você escolheu esse curso é porque, já tem um conhecimento ou experiência sobre as estruturas básicas que compõe um algoritmo ou, já fez o curso “Introdução aos Algoritmos e à Programação Básica com a Linguagem Java” na própria Plataforma +IFMG.

Os algoritmos construídos para rodar na grande maioria dos computadores estão vinculados à arquitetura Von Neuman, ou seja, temos um elemento responsável pelos cálculos que é o **processador** e a **memória**, a qual é incumbida de armazenar os dados e os programas. É importante lembrar que esse último elemento é dividido em células e os algoritmos conseguem acessá-las através das variáveis. Ao aprender programação, percebemos que as variáveis nos atendem muito bem. Através de nomes sugestivos conseguimos consultar e modificar tais espaços na memória.

Contudo, gostaria que imaginássemos um algoritmo que precisasse armazenar o nome das 100 primeiras pessoas a entrar em um evento. A princípio, podemos imaginar a criação de cem variáveis, isto é, nome1, nome2, ..., nome100. Mas essa seria a melhor solução? Imagine o quão trabalhoso seria declarar ou manipular todas essas variáveis. O código fonte ficaria confuso pela quantidade de linhas que seriam demandadas, por exemplo, para a leitura de todos os nomes, teríamos cem linhas do tipo “*nome1 = teclado.next();*”.

Para solucionar esse problema, a maioria das linguagens de programação disponibilizam o conceito de **vetor**. Um vetor é uma lista de dados de um determinado tipo, ou seja, tem-se um nome vinculado à um conjunto de valores. Como definição formal temos que um vetor é uma variável composta, homogênea e unidimensional (DEITEL, 2013). Um vetor também pode ser chamado de “*array*” e nesse tipo de dados, sempre são alocadas posições contínuas de memória do computador para armazenar as várias informações. Veja na Figura 1 como podemos imaginar essa ideia.



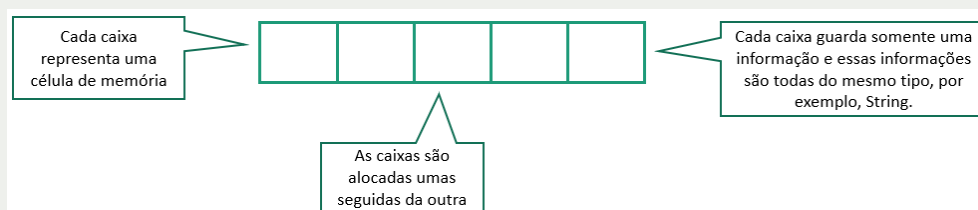


Figura 1 – Representação de vetores na memória.

Fonte: próprio autor.

Para ser mais exato, a Figura 2 mostra como seria alocado, na memória do computador, um vetor chamado “idades”, esse vetor armazena as idades de cinco crianças. Veja nessa representação que cada caixa (posição de memória) recebe um índice. Essa informação é importante, porque através do nome do vetor mais o índice é possível acessar as células de memória, por exemplo, “*idade[0]*” ou “*idade[2]*” representam, respectivamente o valor de 10 anos e 14 anos. Assim, o número que está dentro dos colchetes representa o acesso a uma determinada posição do vetor. Na maioria das vezes, teremos que indicar em qual célula do vetor queremos acessar. É importante ressaltar que esses índices começam do número zero. Então se tivermos um vetor de 5 posições, os índices vão de zero a quatro.

	0	1	2	3	4	← índice
idades	10	12	14	13	9	

Figura 2 – Representação de vetor de idades na memória.

Fonte: próprio autor.

Agora podemos entender mais a definição de Deitel (2013), quando diz que, o vetor é **composto**, pois tem posições de memória contíguas, identificadas por um único nome, individualizadas através de índices. **Homogêneo**, pois todo o conteúdo do vetor é de um mesmo tipo de dados e **unidimensional**, por termos apenas um único índice.



**Mídia digital:** Se você ainda não tem as ferramentas de programação instaladas no seu computador, na sala virtual, assista ao vídeo “Instalando os programas”, o vídeo irá te guiar na instalação do Java e do Netbeans.

Para finalizar essa introdução, a Figura 3 mostra outro exemplo de representação de um vetor chamado “c” com 12 posições de números inteiros. Se criarmos uma linha de código para imprimir o conteúdo do índice 7 teremos o número 62 como resultado.

Nome do array (c) →	c[ 0 ]	-45
	c[ 1 ]	6
	c[ 2 ]	0
	c[ 3 ]	72
	c[ 4 ]	1543
	c[ 5 ]	-89
	c[ 6 ]	0
	c[ 7 ]	62
	c[ 8 ]	-3
	c[ 9 ]	1
Índice (ou subscripto) do elemento no array c →	c[ 10 ]	6453
	c[ 11 ]	78

Figura 3 – Representação de vetor de números inteiros na memória.  
Fonte: Deitel (2013).

## 1.2. Declaração/Instanciação de vetores

Aprendemos que antes de usar uma variável devemos declará-la. Para fazer uso de vetores não é diferente. Nós temos que preparar a memória do computador, mas agora são necessários dois passos básicos:

1. A declaração;
2. A criação (instanciação) na memória.

A Figura 4 mostra um trecho de código demonstrando como pode ser feita a **declaração** de vetores. A declaração não reserva espaço de memória no computador. Ela apenas avisa ao compilador que as variáveis são vetores. O ponto chave aqui são os colchetes. Eles indicam justamente que as variáveis “idades”, “idades1”, “vetor1” e “vetor2” são vetores. Por padrão, os programadores costumam usar nomes de vetores no plural, justamente para indicar que temos mais de um valor referenciado para aquele nome.

```
int idades[];

//ou

int[] idades1;

//-----

double vetor1[], vetor2[];

//ou

double[] vetor11, vetor21;
```

Figura 4 – Declaração de vetores.  
Fonte: próprio autor.

Ainda na Figura 4, repare que os colchetes podem ser colocados depois do tipo da variável ou depois do nome dela, isso é opcional. O Java aceita a declaração de vetores de diversos tipos. Mas, para esse curso vamos fazer uso de vetores dos tipos primitivos de dados, ou seja, *int*, *double*, *float*, *String*, *char*.

No segundo passo, ou seja, na **criação** é alocado espaço na memória do computador para armazenar os dados, isso é feito utilizando a palavra reservada “*new*”, nessa etapa deve-se informar também o tamanho do vetor. A Figura 5 tem um trecho de código demonstrando como pode ser feita a instanciação. Repare que ela deve ser feita depois da declaração. Note que foi criado um vetor para armazenar as idades de cinco pessoas. Já o “vetor1” e “vetor2” têm tamanho dez. Ou seja, irão receber dez números.

A criação é feita do mesmo jeito, independentemente da forma da declaração

```

int idades[];           //declaração
idades = new int[5];    //criação (instanciação)

//ou

int[] idades1;
idades1 = new int[5];

//outro exemplo:

double vetor1[], vetor2[]; //declaração
vetor1 = new double[10];   //criação (instanciação)
vetor2 = new double[10];   //criação (instanciação)

```

Figura 5 – Instanciação de vetores.  
Fonte: próprio autor.

No momento da instanciação do vetor na memória, o Java insere valores padrões em cada célula do vetor. Isso ocorre de acordo com tipo de dados que o vetor irá armazenar. Veja na Figura 6 alguns exemplos. Quando o vetor é de inteiros, o valor padrão é o zero. Quando o vetor é de *strings*, o valor padrão é a palavra reservada “*null*”, a qual indica que não existe valor/texto naquela célula. Para *double* e *float* o valor padrão é o “0.0”.

<code>int idades[] = new int[5];</code>	0	1	2	3	4
	0	0	0	0	0
<code>String[] nomes = new String[5];</code>	0	1	2	3	4
	null	null	null	null	null
<code>double vetor1[] = new double[10];</code>	0	1	2	3	4
	0.0	0.0	0.0	0.0	0.0

Figura 6 – Valor padrão dos vetores.  
Fonte: próprio autor.

Existe uma forma de declarar, alocar memória e preencher o vetor ao mesmo tempo. Esse tipo de situação é útil quando já temos os dados que precisamos. A Figura 7 mostra um exemplo, nesse caso, o vetor de inteiros chamado “n” tem 5 posições. Se imprimirmos o código “n[4]” teremos o retorno do valor 99.

```
int n[] = {10, 20, 65, 5, 99};
```

Figura 7 – Declaração e instanciação de vetores.  
Fonte: próprio autor.



**Atenção:** Antes de fazer uso de um vetor, lembre-se de **declarar e instanciar** essa nova estrutura de dados. Caso contrário, o Java apontará o erro do tipo “*NullPointerException*”.



**Mídia digital:** Antes de prosseguirmos, assista também ao vídeo “Introdução aos vetores” para fazer uma revisão de tudo o que apresentado até agora.

### 1.3. Atribuição de valores e leitura de dados

As operações mais corriqueiras, ao se trabalhar com vetores, são a atribuição de valores e a leitura de dados. Para essas duas operações, você deve ter em mente que devemos dizer qual o nome de vetor e qual o índice do vetor queremos acessar. Imagine um carteiro entregando correspondências. Ele precisa do nome da rua e do número da casa. Para os vetores não é diferente. Você precisa do nome do vetor e do número do índice. A Figura 7 mostra um exemplo de como atribuir valores a um vetor de *strings*. Na imagem temos o código fonte e uma representação gráfica do que acontece. Primeiro o vetor chamado “alunos” é declarado e criado na memória (tamanho de cinco posições). Logo em seguida, atribuímos valores para quatro posições diferentes no vetor. Repare que na posição de índice “um” foi atribuído uma *string* vazia. Como não mexemos na posição de índice 3, essa célula continua com o valor padrão, ou seja, o valor “null”.

```
String alunos[] = new String[5];
alunos[0] = "Mary";
alunos[1] = "";
alunos[2] = "Thomas";
alunos[4] = "Jhony";
```

	0	1	2	3	4
alunos	Mary		Thomas	null	Jhony

Figura 7 – Exemplo de atribuição de valores em um vetor.

Fonte: próprio autor.

Para acessar/ler um valor em uma posição de um vetor, use sempre o nome do vetor e o índice desejado entre colchetes. Veja um trecho de código e o resultado da execução dele na Figura 8. Estamos usando o mesmo vetor “alunos” da figura anterior. As duas primeiras linhas de código imprimem, respectivamente, a primeira e a última posição do vetor. Veja o resultado no “Console”. O terceiro comando de impressão (*System.out.println*) faz uso de uma variável auxiliar chamada “melhorAluno”. Essa variável recebe o conteúdo da célula de índice dois. Já o quarto e último comando de impressão, usa uma variável do tipo inteira para indicar o índice que o programa irá imprimir. Nesse caso o índice será o de número dois.

```
System.out.println(alunos[0]);

System.out.println(alunos[4]);

String melhorAluno = alunos[2];
System.out.println("Melhor aluno: "+ melhorAluno);

int posição = 2;
System.out.println(alunos[posição]);
```

```
Console
<terminated> Exemplo1 [Java Application] C:\Program Files\Java\j...
Mary
Jhony
Melhor aluno: Thomas
Thomas
```

Figura 8 – Exemplo de leitura de valores em um vetor.

Fonte: próprio autor.

Para finalizar essa seção, vamos analisar alguns erros que são muito comuns e que devemos evitar. Mas se você entendeu o que conversamos até o momento, você não irá cometê-los. A Figura 9 mostra alguns trechos de código. A primeira linha apenas declara e instancia um vetor chamado “alunos” com cinco posições, ou seja, temos os índices de zero a quatro. A segunda linha está com erro, porque não indicamos em qual posição/índice a “Janes” será inserida. A terceira linha tenta imprimir o vetor, mas quando executamos esse tipo comando, o Java imprime a posição de memória inicial, ao invés de

imprimir o conteúdo, isso porque não informamos o índice. Para imprimir o conteúdo de todo o vetor, o programa terá que acessar posição por posição. Já a última linha, força um erro muito comum, o qual é o de acessar um índice que não existe no vetor. Sabemos que esse vetor tem o índice inicial zero e o final com o valor quatro. Nesse caso, estamos tentando acessar uma posição inexistente.

```
String alunos[] = new String[5];

alunos = "Janes";           ← Erro

System.out.println(alunos); ← Imprime a posição de memória e não os dados.

alunos[400] = "Jhony";      ← Erro
```

Figura 9 – Exemplo de erros comuns.  
Fonte: próprio autor.



**Atenção:** Sempre informe o nome do vetor e o índice que você deseja acessar, só assim você conseguirá recuperar ou alterar uma informação em um vetor.

## 1.4. Percorrendo o vetor

É muito comum criarmos algoritmos que precisam percorrer todo o vetor, ou seja, acessar todas as posições de um vetor. Por exemplo, logo no início do programa pode ser necessário cadastrar duzentas pessoas em um *array*, ou, criar um algoritmo para descobrir se uma determinada pessoa está inserida em um vetor. Um terceiro exemplo seria, somar a idade de dez pessoas inseridas em um vetor. Esses são apenas três exemplos de milhares de situações em que seria útil percorrer uma estrutura de dados sequencial.

A primeira forma de acessar todas as posições de um vetor é digitando o nome do vetor e o índice desejado, então basta repetir essa linha a quantidade de vezes necessária, mudando apenas o número do índice. Contudo, sabe-se que repetir uma linha de código dezenas ou centenas de vezes não é uma solução elegante. A alternativa mais indicada seria utilizar os laços de repetição. A Figura 10 mostra um exemplo. A ideia do código é ler cinco valores do tipo inteiro e colocar esses valores em cada posição do vetor. As primeiras duas linhas criam, respectivamente, um objeto para leitura de dados e o vetor chamado “x”.

A Figura 10 usa do laço de repetição “for” para percorrer as cinco posições. Lembre-se que esse laço de repetição usa uma variável de controle, nesse caso o nome dela é “i”. Essa variável começa com zero e será incrementada até alcançar o limite, que nesse caso é o quatro, pois quando a variável “i” alcançar o valor de 5, o laço não será mais executado. O grande segredo está na linha iniciada com “x[i]”. A cada iteração do laço, a variável “i” recebe um valor, então estaremos acessando a posição que essa variável

indicar. Se a variável tiver o valor 3, estaremos acessando a posição do vetor de índice três. Ou seja, a quarta posição, pois sabe-se que os vetores começam do índice zero.

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

double x[] = new double[5];

for (int i = 0; i < 5; i++){

    System.out.printf("Digite o %dº número", i+1);
    x[i] = teclado.nextFloat();
}
```

O valor de "i" vai variar e assim podemos acessar todas as posições do vetor.

Como o índice começa do zero somamos 1 para mostrar o usuário os valores de 1 a 5

Figura 10 – Exemplo de laço para percorrer um vetor.

Fonte: próprio autor.

Veja uma ilustração do laço de repetição sendo executado na Figura 11. Cada linha dessa figura indica uma iteração do comando "for". Na primeira execução, a variável "i" tem o valor zero. O usuário entrou com o valor hipotético 95. Então o vetor "x" recebe o valor de 95 na posição de índice zero. Na segunda iteração ("i" igual a 1), o usuário insere o valor 13. Assim, a segunda célula do vetor terá esse valor. Esse mesmo comportamento é repetido até a condição do laço ser falsa. Ou seja, quando "i" tem um valor maior ou igual a cinco.

MEMÓRIA						TELA
i = 0	X	95				Digite o 1º número 95
		0	1	2	3	4
i = 1	X	95	13			Digite o 2º número 13
		0	1	2	3	4
i = 2	X	95	13	-25		Digite o 3º número -25
		0	1	2	3	4
i = 3	X	95	13	-25	47	Digite o 4º número 47
		0	1	2	3	4
i = 4	X	95	13	-25	47	Digite o 5º número 0
		0	1	2	3	4

Figura 11 – Ilustração da iteração do vetor pelo comando "for".

Fonte: ASCENCIO (2088).

A Figura 12 mostra um exemplo simples de inicialização de um vetor. Ou seja, o trecho de código percorre todo o vetor inserindo o valor zero em cada posição. Isso é muito útil para colocarmos um valor desejado em cada posição da estrutura de dados que estamos trabalhando.

```
double notas[] = new double[50];

for (int i = 0; i < 50; i++){

    notas[i] = 0;
}
```

Figura 12 – Segundo exemplo de laço para percorrer um vetor.

Fonte: próprio autor.

A Figura 13 é uma continuação da figura anterior. O trecho de algoritmo dessa figura percorre todo o vetor para imprimir o conteúdo na tela. Como nos outros exemplos, a variável “i” assumirá valores diferentes em cada iteração, ou seja, teremos o “i” variando de zero a quarenta e nove. Quando o “i” assumir o valor de 50, a condição de execução será falsa e o laço será encerrado.

```
for (int i = 0; i < 50; i++){

    System.out.println(notas[i]);
}
```

Figura 13 – Terceiro exemplo de laço para percorrer um vetor.

Fonte: próprio autor.



**Mídia digital:** Antes de prosseguirmos, assista ao vídeo “Percorrendo os vetores” para fazer uma revisão dessa seção.

## 1.5. Descobrindo o tamanho do vetor

Os vetores **não** são dinâmicos, ou seja, uma vez definido o tamanho, não é permitido aumentar ou diminuir o comprimento dele. Contudo podemos definir seu tamanho em tempo de execução do programa. Isso é muito útil, porque em muitas das vezes, só conhecemos o tamanho do vetor depois de obter outra informação. A Figura 14 mostra um exemplo de definição do tamanho do vetor em tempo de execução. Quando o algoritmo é executado, o usuário é requisitado a informar a quantidade de pessoas. Assim, saberemos o tamanho do vetor que precisamos criar. Por fim, esse tamanho é armazenado na variável “qtde”, a qual é utilizada pelo compilador *Java* na alocação de memória do vetor.



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

System.out.println("Quantas pessoas serão cadastradas?");
int qtde = dados.nextInt();

String pessoas[] = new String[qtde];
```

O tamanho do vetor dependerá do valor informado pelo usuário.

Figura 14 – Definindo o tamanho do vetor em tempo de execução.

Fonte: próprio autor.

É importante ressaltar que a variável “qtde”, mostrada na Figura 14, pode não estar acessível em todo o programa, ou o conteúdo dessa variável pode ser alterado em algum outro ponto do algoritmo. Essa informação é útil, por exemplo, para definir a condição de parada do laço de repetição. Então como descobrir posteriormente o tamanho do vetor? A resposta é simples. O *Java* disponibiliza uma propriedade em todo vetor chamada ***length***. Se traduzirmos essa palavra para o português, ela quer dizer justamente, comprimento. Para buscar essa informação, basta usar o nome do vetor, o sinal de ponto final e a palavra reservada *length*.

A Figura 15 mostra um exemplo de uso. Esse trecho de algoritmo é continuação da figura anterior. Vamos imaginar que não sabemos, de ante mão, qual o tamanho do vetor. Usamos então o comando “*pessoas.length*” para obter o tamanho do vetor. Assim, conseguimos criar uma condição de execução do laço. Ou seja, enquanto a variável de controle “*i*”, for menor que o tamanho do vetor, o laço de repetição continuará executando. Quando a variável de controle alcançar o tamanho do vetor, o laço é encerrado. Ainda nesse trecho de algoritmo, dentro do laço, preencheremos todas as células com nomes de pessoas, os quais são informados pelo usuário.

```
for (int i = 0; i < pessoas.length; i++){

    System.out.println("Qual o nome "+i);
    pessoas[i] = dados.next();
}
```

Retorna o tamanho do vetor.

Figura 15 – Exemplo de uso da propriedade “length”.

Fonte: próprio autor.



**Dica do Professor:** É uma boa prática usar o *length* para se referenciar ao tamanho do vetor, mesmo quando conhecemos o tamanho do vetor, isso facilita futuras alterações no código.

## 1.6. Exemplos

Essa seção apresenta três exemplos de uso de vetores. O objeto é reforçar o conteúdo aprendido até o momento. Para facilitar o entendimento, numeramos as linhas dos algoritmos e reforçamos que esses códigos foram digitados dentro de um método *main* de uma classe *Java* qualquer. A Figura 16 mostra o primeiro exemplo, a ideia aqui é criar um algoritmo que recebe 20 números reais e imprime somente os pares. A linha 1 cria o objeto responsável pela leitura de dados. A linha 3 declara e instancia um vetor do tipo *double* com vinte posições, ou seja, teremos os índices zero ao dezenove. Para preencher todo o vetor usamos um laço de repetição e fazemos a leitura dos dados (linhas 5 a 8). No último laço de repetição (linhas 10 a 14), percorremos todas as células e verificamos se o número é par ou ímpar. Para isso usamos o operador “%”. Ele é responsável por pegar o resto da divisão e essa divisão é de números inteiros, ou seja, não tem casa decimal. Se dividirmos o número por dois e o resto for zero, então o número é par (linha 12). A linha 13 imprime o número se a condição for verdadeira.

```

1 Scanner dados = new Scanner(System.in);
2
3 double numeros[] = new double[20];
4
5 for (int i = 0; i < numeros.length; i++){
6     System.out.printf("Digite o %dº número", i+1);
7     numeros[i] = dados.nextInt();
8 }
9
10 for (int i = 0; i < numeros.length; i++){
11
12     if (numeros[i] % 2 == 0)
13         System.out.println(numeros[i]);
14 }

```

Figura 16 – Exemplo de algoritmo para impressão de números pares.

Fonte: próprio autor.

A Figura 17 apresenta um exemplo de algoritmo que armazene 20 números em um vetor chamado “valores” e depois imprime qual é o maior valor armazenado. A linha 1 cria o objeto responsável pela leitura de dados. As linhas 2 e 3 são responsáveis, respectivamente, por declarar e instanciar um vetor do tipo *int* com vinte posições. Para preencher todo o vetor, foi usado um laço de repetição, nele requisitamos a entrada do usuário para todas as células do vetor (linhas 4 a 7). A grande novidade acontece da linha 9 a 15. Primeiro foi criada uma variável chamada “auxiliar”. Ela foi inicializada com o menor valor inteiro possível no *Java*. Isso vai garantir que qualquer número do vetor seja maior

que esse valor inicial. No segundo laço (linha 10), verificamos se a posição atual, apontada pela variável “i” tem um valor maior que o da variável chamada “auxiliar”, se for verdade, substituímos o valor atual da variável “auxiliar” pelo valor que está na célula atual do vetor. O algoritmo faz essa verificação para todas as posições. No final do laço, o maior valor é impresso na tela (linha 15).

```

1  Scanner dados = new Scanner(System.in);
2  int valores[];
3  valores = new int[20];
4  for (int i = 0; i < valores.length; i++){
5      System.out.printf("Digite o número", i+1);
6      valores[i] = dados.nextInt();
7  }
8
9  int auxiliar = Integer.MIN_VALUE;
10 for (int i = 0; i < valores.length; i++){
11
12     if (valores[i] > auxiliar)
13         auxiliar = valores[i];
14 }
15 System.out.println(auxiliar);

```

Figura 17 – Exemplo de algoritmo para descobrir o maior valor do vetor.

Fonte: próprio autor.

O terceiro e último exemplo de código fonte é exibido na Figura 18, esse algoritmo recebe 10 números inteiros e depois imprime-os na ordem inversa. A linha 1 cria o objeto responsável pela leitura de dados. As linhas 3 e 4 são responsáveis, respectivamente, por declarar e instanciar um vetor do tipo *int* com dez posições. Para preencher todo o vetor, foi usado um laço de repetição, nele requisitamos a entrada do usuário para todas as células do vetor (linhas 6 a 9). A lógica de imprimir os dados invertidos acontece da linha 11 a 14. Primeiro é importante ressaltar que não vamos modificar o vetor e sim a forma de acessá-lo. A linha 11 é onde devemos analisar com mais detalhes. Repare que a variável “i” foi iniciada com o comprimento do vetor menos um. Isso significa que a variável em questão receberá o valor nove, ou seja, o resultado da conta: dez menos um. Repare então que vamos acessar de início a célula com o índice 9 (última posição). A condição de execução está diferente do habitual, isto é, enquanto o “i” for maior ou igual a zero, o laço irá roda, assim conseguiremos alcançar a primeira posição. Ainda nessa linha, tem-se um detalhe importante, ao invés de incrementar, agora estamos decrementando a variável “i”, assim, em cada iteração o valor dessa variável vai diminuir de um.

```

1 Scanner dados = new Scanner(System.in);
2
3 int inverter[];
4 inverter = new int[10];
5
6 for (int i = 0; i < inverter.length; i++){
7     System.out.printf("Digite o número", i+1);
8     inverter[i] = dados.nextInt();
9 }
10
11 for (int i = inverter.length-1; i >=0 ; i--){
12
13     System.out.println(inverter[i]);
14 }

```

Figura 18 – Exemplo de algoritmo para descobrir o maior valor do vetor.

Fonte: próprio autor.



**Mídia digital:** Antes de prosseguirmos, assista ao vídeo “Exemplos de uso de vetores” para fazer uma revisão de tudo o que foi apresentado nessa seção.



**Atenção:** Os vetores sempre tem o primeiro índice com o valor zero. Assim, o último índice tem o valor definido pelo tamanho do vetor menos um.



**Dica do Professor:** O aprendizado de algoritmos é contínuo. Um conceito novo depende de outros já apresentados. Assim, antes de avançar para a próxima semana, tenha certeza que entendeu os conceitos apresentados até aqui e que assistiu todos os vídeos e instalou os softwares requisitados.

Concluída essa semana de estudos é hora de uma pausa para a reflexão. Faça a leitura (ou releitura) de tudo que lhe foi sugerido, assista aos vídeos propostos e analise todas essas informações. Esse intervalo é importante para amadurecer as novas concepções que esta etapa lhe apresentou!

Nos encontramos na próxima semana.

Bons estudos!



### Objetivos

Nessa segunda semana você conhecerá os conceitos básicos sobre matrizes. Vamos manipular essa estrutura para gerenciar informações em formato bidimensional.

### 2.1 Considerações iniciais

Nessa segunda semana, nós vamos evoluir um pouco mais sobre o conhecimento das estruturas de dados. Já aprendemos sobre a estrutura de dados sequencial chamada vetor. Essa é uma ótima solução para abstrair problemas em que temos uma ordem linear entre as informações. Um exemplo disto seria um consultório, as pessoas na sala de espera estão sentadas em qualquer lugar, porém sabe-se quem é o próximo a ser atendido, e o seguinte, e assim por diante.

Imagine agora que você precise criar um algoritmo para armazenar os dados em formato de tabelas, por exemplo, o rendimento diário de um investimento em um determinado mês, uma tabela de fatos da matemática ou os *pixels* de uma imagem a ser processada. Será que várias variáveis ou mesmo vários vetores seriam os mais indicados? Imagine o quão trabalhoso seria declarar ou manipular todas essas variáveis. O código fonte ficaria confuso pela quantidade de linhas que seriam demandadas.



Figura 19 – Ilustração de dados em tabelas/matrizes.

Fonte: <https://unsplash.com/photos/oNlzGALKDwo> acessado em 24/04/2022.

Para solucionar esse problema, a maioria das linguagens de programação disponibilizam o conceito de **matriz**. Uma matriz, assim como um vetor, é um conjunto de dados de um determinado tipo, ou seja, tem-se um nome vinculado a vários valores. Como

definição formal, temos que uma matriz é uma variável composta, homogênea e multidimensional (DEITEL, 2013). Uma matriz também pode ser chamada de *array* multidimensional, e nesse tipo de dados, assim como vetores, nem sempre são alocadas posições contínuas de memória do computador, mas as linguagens de programação conseguem abstrair o acesso de forma contínua utilizando índices para as várias dimensões. Veja na Figura 20 como podemos imaginar essa ideia, essa figura mostra uma matriz de duas dimensões e outra com três.

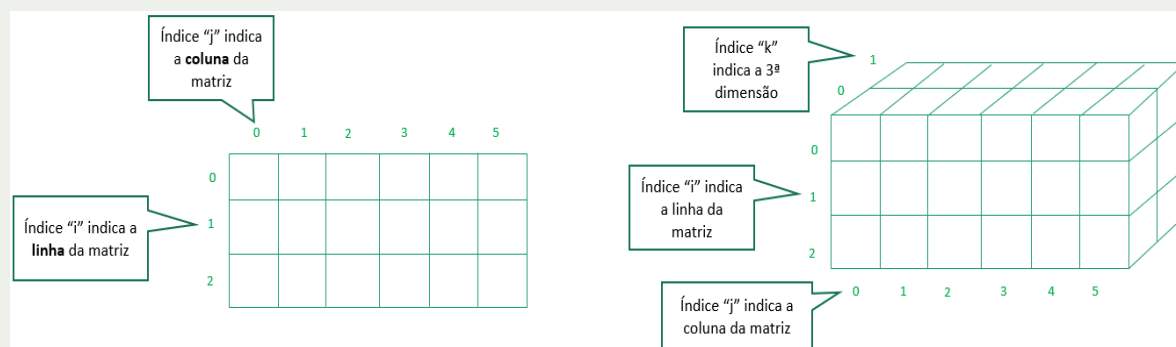


Figura 20 – Representação de matrizes na memória.

Fonte: próprio autor.

O que distingue as matrizes de variáveis simples são os índices que referenciam sua localização dentro da estrutura. Uma variável do tipo matriz precisa de um índice para cada uma de suas dimensões de valores. Assim, fica fácil perceber que as matrizes se diferem de vetores porque têm mais de um índice. Veja a Figura 21 para mais alguns detalhes.

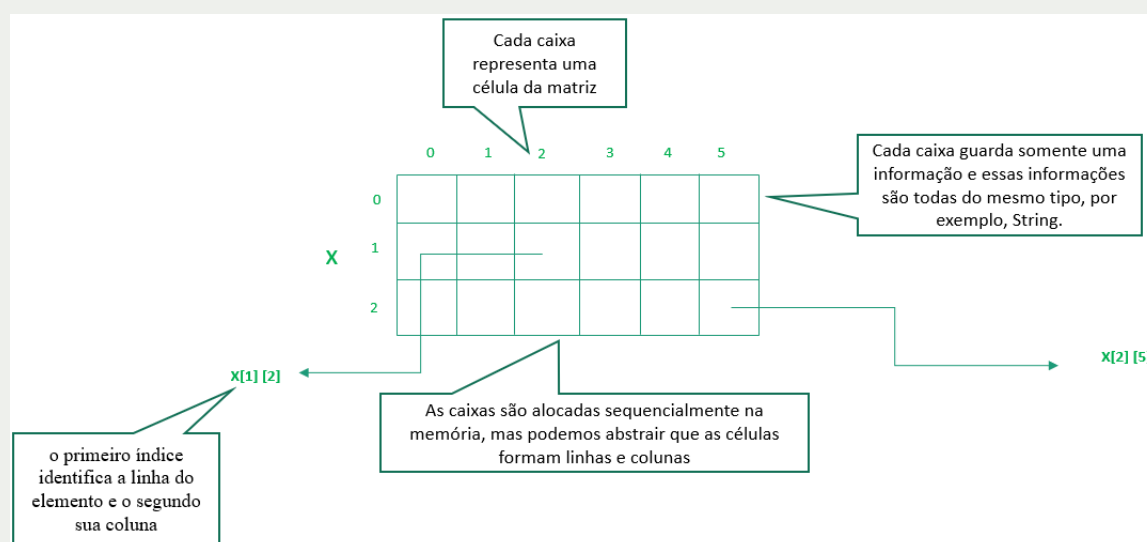


Figura 21 – Detalhes da representação matricial na memória.

Fonte: próprio autor.

Na Figura 21 fica fácil perceber que existem semelhanças com vetores, por exemplo, os índices começam do zero, cada célula armazena uma informação. A principal

diferença é que temos que acessar as células indicando os índices das várias dimensões. Por exemplo, usamos o comando “x[1][2]” para acessar a célula da linha um e coluna dois de uma matriz chamada “x”. Usamos um par de abre e fecha colchetes para cada dimensão. Para reforçar a ideia, veja os endereços possíveis em uma matriz de três linhas e quatro colunas na Figura 22.

	Coluna 0	Coluna 1	Coluna 2	Coluna 3
Linha 0	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
Linha 1	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
Linha 2	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

Índice de coluna  
Índice de linha  
Nome do array

Figura 22 – Representação de acesso de células em uma matriz.  
Fonte: Deitel (2013).

## 2.2 Declaração/Instanciação de matrizes

Aprendemos que antes de usar uma variável devemos declará-la. Para fazer uso de matrizes não é diferente. Nós temos que preparar a memória do computador, mas agora são necessários dois passos básicos:

1. A declaração;
2. A criação (instanciação) na memória.

A Figura 23 mostra um trecho de código demonstrando como pode ser feita a **declaração** de matrizes. A declaração não reserva espaço de memória no computador. Ela apenas avisa ao compilador que as variáveis são matrizes. O ponto chave aqui são os dois pares de colchetes. Eles indicam justamente que as variáveis “idades”, “idades1”, “matriz1” e “matriz2” são matrizes. Por padrão, os programadores costumam usar nomes de matrizes no plural, justamente para indicar que temos mais de um valor referenciado para aquele nome.



```

int idades[][];

//ou

int[][] idades1;

//-----

double matriz1[][], matriz2[][];

//ou

double[][] matriz11, matriz21;

```

Figura 23 – Declaração de matrizes.  
Fonte: próprio autor.

Ainda na Figura 23, repare que os pares de colchetes podem ser colocados depois do tipo da variável ou depois do nome dela, isso é opcional. O Java aceita a declaração de matrizes de diversos tipos. Mas, para esse curso vamos fazer uso de dos tipos primitivos de dados, ou seja, *int*, *double*, *float*, *String* e *char*.

No segundo passo, ou seja, na **criação** da matriz será alocado espaço na memória do computador para armazenar os dados, isso é feito utilizando a palavra reservada “*new*”, nessa etapa deve-se informar também o tamanho da matriz. A Figura 24 tem um trecho de código demonstrando como pode ser feita a instanciação. Repare que ela deve ser feita depois da declaração. Note que foi criado uma matriz para armazenar as idades de vinte pessoas dispostas em 5 linhas de 4 colunas. Já a “matriz1” e “matriz2” têm tamanho 100, ou seja, irão receber dados do tipo “double” em dez linhas de dez colunas cada.

A criação é feita do mesmo jeito, independentemente da forma da declaração

```

int idades[][]; //declaração
idades = new int[5][4]; //criação (instanciação)

//ou

int[][] idades1;
idades1 = new int[5][4];

//outro exemplo:

double matriz1[][], matriz2[][]; //declaração
matriz1 = new double[10][10]; //criação (instanciação)
matriz2 = new double[10][10]; //criação (instanciação)

```

Figura 24 – Instanciação de matrizes.  
Fonte: próprio autor.

A Figura 25 mostra dois pontos interessantes. O primeiro é que se pode declarar e instanciar uma matriz em uma mesma linha de código e o segundo ponto é sobre os valores padrões, o Java insere valores em cada célula logo depois da instanciação. Isso

ocorre de acordo com tipo de dados que a matriz irá armazenar. Quando a matriz é de inteiros, o valor padrão é o zero. Quando a matriz é de *strings*, o valor padrão é a palavra reservada “*null*”, a qual indica que não existe valor/texto naquela célula. Para *double* e *float* o valor padrão é o “0.0”.

```
int idades[][] = new int[2][5];
```

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0

```
String[][] nomes = new String[2][5];
```

	0	1	2	3	4
0	null	null	null	null	null
1	null	null	null	null	null

```
double matriz1[][] = new double[2][5];
```

	0	1	2	3	4
0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0

Figura 25 – Declaração e instanciação de matrizes e seus valores padrões.

Fonte: próprio autor.

Existe uma forma de declarar, alocar memória e preencher a matriz ao mesmo tempo. Esse tipo de situação é útil quando já temos os dados que precisamos. O trecho de código abaixo é um exemplo, nesse caso, a estrutura de inteiros chamada “n” tem 3 linhas com duas colunas cada. Se imprimirmos o código “n[1,1]” teremos o retorno do valor 5.

```
int n[][] = {{10,20},{65,5},{6,15}};
```

Não é o foco desse curso, mas vale mencionar que é possível criar matrizes irregulares em Java. Uma matriz irregular é aquela que pode ter linhas com comprimentos diferentes, ou seja, com quantidades de colunas diferentes. A Figura 26 mostra dois exemplos, aonde podemos ver o código responsável pela criação das matrizes e a representação gráfica de como seria o formato das mesmas.

```
int m[][] = { {10}, {65, 5}, {6, 1, 5} };
```

	0	1	2
0	10		
1	65	5	
2	6	1	5

```
int k[][] = new int[2][];
k[0] = new int[5];
k[1] = new int[3];
k[1][1] = 45
```

	0	1	2	3	4
0					
1		45			

Figura 26 – Declaração e instanciamento de matrizes irregulares.

Fonte: próprio autor.



**Atenção:** Antes de fazer uso de uma matriz, lembre-se de **declarar e instanciar** essa nova estrutura de dados. Caso contrário, o Java apontará o erro do tipo “*NullPointerException*”.



**Mídia digital:** Antes de prosseguirmos, assista também ao vídeo “Introdução às matrizes” para fazer uma revisão de tudo o que apresentado até agora.

## 2.3 Atribuição de valores e leitura de dados

As operações mais corriqueiras, ao se trabalhar com matrizes, são a atribuição de valores e a leitura de dados. Para essas duas operações, você deve ter em mente que devemos dizer qual o nome da matriz e qual a célula utilizando os índices necessários para identificação única da posição. A Figura 27 mostra um exemplo de como atribuir valores a uma matriz de *strings*. Na imagem temos o código fonte e uma representação gráfica do que acontece. Primeiro a matriz chamada “alunos” é declarada e criada na memória (tamanho de 10 posições – 2 linhas e 5 colunas). Logo em seguida, atribuímos valores para quatro posições diferentes na matriz. Repare que na posição de linha “zero” e coluna “dois” foi atribuído uma *string* vazia. Como não mexemos nas demais células, elas continuam com o valor padrão, ou seja, o valor “null”.

```
String alunos[] = new String[2][5];
alunos[0][0] = "Mary";
alunos[0][2] = "";
alunos[1][2] = "Thomas";
alunos[0][4] = "Jhony";
```

	0	1	2	3	4	← índice
alunos 0	Mary	null		null	Jhony	
1	null	null	Thomas	null	null	

Figura 27 – Exemplo de atribuição de valores em uma matriz.  
Fonte: próprio autor.


Para acessar/ler um valor em uma posição de uma matriz, use sempre o nome da matriz e os índices desejados entre os dois colchetes. Veja um trecho de código e o resultado da execução dele na Figura 28. Estamos usando o mesmo vetor “alunos” da figura anterior. As duas primeiras linhas de código imprimem, respectivamente, a primeira e a última coluna da linha superior da matriz. Veja o resultado no “Console”. O terceiro comando de impressão (*System.out.println*) usa duas variáveis do tipo inteiro para indicar, respectivamente, a linha e coluna da célula que será impressa. Já o quarto e último comando de impressão, faz uso de uma variável auxiliar chamada “melhorAluno”. Essa variável recebe o conteúdo da célula de linha 1 e coluna 2.

```
System.out.println(alunos[0][0]);

System.out.println(alunos[0][4]);

int linha = 1;
int coluna = 2;
System.out.println(alunos[linha][coluna]);

String melhorAluno = alunos[1][2];
System.out.println("Melhor aluno: " + melhorAluno);
```



```
<terminated> Exemplo1 [Java Application] C:\Program Files\Java\jn
Mary
Jhony
Thomas
Melhor aluno: Thomas
```

Figura 28 – Exemplo de leitura de valores em uma matriz.  
Fonte: próprio autor.



**Atenção:** Sempre informe o nome da matriz e os índices de acordo com cada dimensão para acessar uma célula. Verifique sempre se os índices estão respeitando o comprimento das linhas e colunas da estrutura de dados instanciada.

## 2.4 Percorrendo a matriz

É muito comum criarmos algoritmos que precisam percorrer toda a matriz, ou seja, acessar todas as posições da estrutura de dados. Por exemplo, ao acionar um determinado botão, pode ser necessário apagar todos os dados de um *array multidimensional*, ou, criar um algoritmo para descobrir se uma imagem armazenada em uma matriz tem uma determinada cor de pixel. Um terceiro exemplo seria, somar todos os números de uma matriz com os valores de rendimento diário, a qual representa os dias de um mês. Esses são apenas três exemplos de milhares de situações em que seria útil percorrer uma estrutura de dados multidimensional.

A primeira forma de acessar todas as posições de uma matriz é digitando o nome dela e os índices desejados, então basta repetir essa linha a quantidade de vezes necessária, mudando apenas os números dos índices. Contudo, sabe-se que repetir uma linha de código dezenas ou centenas de vezes não é uma solução elegante. A alternativa mais indicada seria utilizar os laços de repetição. A ideia agora é usar um laço de repetição por dimensão da matriz. Vamos usar um laço para percorrer as diferentes linhas e dentro da linha atual teremos outra estrutura de repetição para percorrer as colunas. A Figura 29 mostra, usando setas, o sentido de caminhamento nas células em uma matriz.

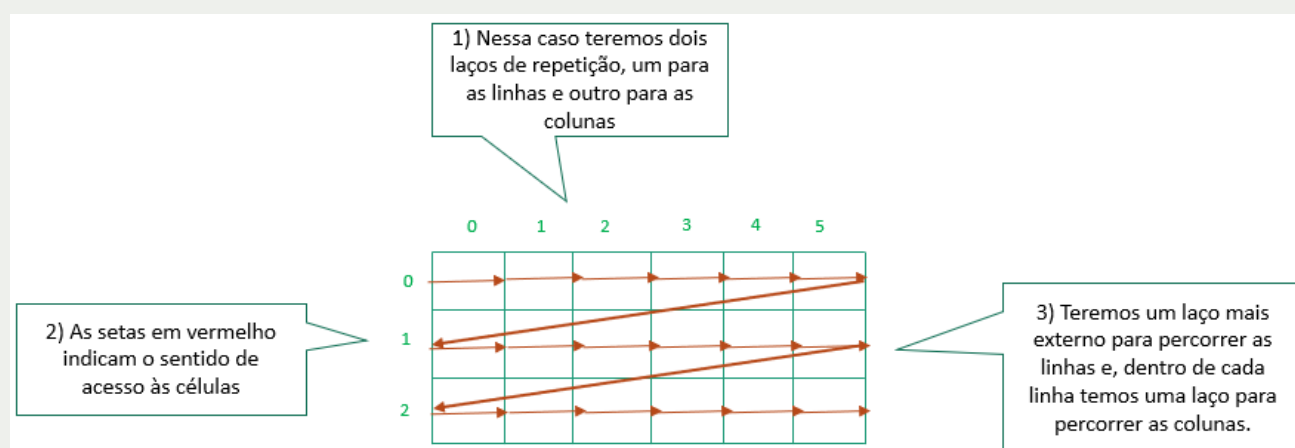


Figura 29 – Exemplo de caminhamento em uma matriz.

Fonte: próprio autor.

A Figura 30 exibe um exemplo, usando o laço de repetição “*for*”, para mostrar o caminhamento na matriz, afim de ser mais exato, o código usa dois comandos “*for*”. A ideia é percorrer as quinze posições de uma matriz. Lembre-se que o laço de repetição usa uma variável de controle, nesse caso o nome da primeira variável é “linha” e a do laço mais interno é “coluna”. Essas variáveis começam com zero e serão incrementadas até alcançarem seus limites/tamanhos. Para cada iteração no laço mais externo, haverá cinco iterações no laço mais interno.

1) Usamos variáveis para definir o tamanho da matriz antes da sua declaração e instanciação

```
int quantidadeLinhas = 3;
int quantidadeColunas = 5;
int matriz[][] = new int[quantidadeLinhas][quantidadeColunas];
```

2) O primeiro laço de repetição manipula a variável da linha e o segundo a coluna.

```
for (int linha = 0; linha < quantidadeLinhas; linha++){

    for(int coluna = 0; coluna < quantidadeColunas; coluna++){

        //aqui fazemos alguma coisa com a célula atual:
        //                                matriz[linha][coluna]

    }

}
```

Figura 30 – Exemplo de uso de laços de repetição para percorrer uma matriz.  
Fonte: próprio autor.

Veja uma ilustração dos laços de repetição sendo executados na Figura 31. Cada linha dessa figura indica uma iteração do primeiro e do segundo comando “for”. Na primeira execução, as variáveis “linha” e “coluna” têm os valores zero. O usuário entrou com o valor hipotético 12. Então a variável “matriz” recebe o valor de 12 na primeira posição do *array* multidimensional. Na segunda iteração (“linha” igual a zero e coluna igual a um), o usuário insere o valor 9. Assim, a segunda célula da primeira linha terá esse valor. Esse mesmo comportamento é repetido até as condições dos dois laços sejam falsas. Ou seja, quando “linha” tem um valor maior ou igual a 3 e coluna maior ou igual a 5.

MEMÓRIA		TELA
linha	coluna	
0	0	Digite o número da linha 0 e coluna 0: 12
	1	Digite o número da linha 0 e coluna 1: 9
	2	Digite o número da linha 0 e coluna 2: 3
	3	Digite o número da linha 0 e coluna 3: 7
	4	Digite o número da linha 0 e coluna 4: -23
1	0	Digite o número da linha 1 e coluna 0: 15
	1	Digite o número da linha 1 e coluna 1: 4
	2	Digite o número da linha 1 e coluna 2: 2
	3	Digite o número da linha 1 e coluna 3: 34
	4	Digite o número da linha 1 e coluna 4: -4
2	0	Digite o número da linha 2 e coluna 0: 3
	1	Digite o número da linha 2 e coluna 1: 45
	2	Digite o número da linha 2 e coluna 2: 3
	3	Digite o número da linha 2 e coluna 3: 0
	4	Digite o número da linha 2 e coluna 4: -3

Figura 31 – Ilustração da iteração da matriz usando dois comandos “for”.  
Fonte: Adaptado de ASCENCIO (2088).

A Figura 32 mostra um exemplo simples de inicialização de matriz. Ou seja, o trecho de código percorre todo o *array* multidimensional inserindo o valor zero em cada posição. Isso é muito útil para colocarmos um valor desejado nesse tipo de estrutura de dados.

```
double notas[][] = new double[10][5];

for (int i = 0; i < 10; i++){
    for (int j = 0; j < 5; j++){
        notas[i][j] = 0;
    }
}
```

A variável "i" assumirá os valores de 0 a 9 para acessar as linhas e, a variável "j" receberá os valores entre 0 e 4 que representam as colunas. Para todas as células da matriz inserimos o valor zero.

Figura 32 – Segundo exemplo de laço para percorrer um vetor.  
Fonte: próprio autor.

A Figura 33 é uma continuação da figura anterior. O trecho de algoritmo dessa figura percorre toda matriz para imprimir o conteúdo na tela. Como no outro exemplo, as variáveis "i" e "j" assumirão valores diferentes em cada iteração, ou seja, teremos o "i" variando de zero a nove e "j" de zero a quatro. Para cada uma dessas combinações de valores, o algoritmo imprime uma célula da matriz.

```
for (int i = 0; i < 10; i++){
    for (int j = 0; j < 5; j++){
        System.out.println(notas[i][j]);
    }
}
```

Figura 33 – Terceiro exemplo para percorrer uma matriz.  
Fonte: próprio autor.



**Mídia digital:** Antes de prosseguirmos, assista ao vídeo "Percorrendo as matrizes" para fazer uma revisão dessa seção.

## 2.5 Descobrindo o tamanho da matriz

Assim como os vetores, as matrizes **não** são dinâmicas, ou seja, uma vez definido o tamanho, não é permitido aumentar ou diminuir o comprimento dessas estruturas. Contudo podemos definir seu tamanho em tempo de execução do programa. Isso é muito útil, porque em muitas das vezes, só conhecemos o tamanho da matriz depois de obter outra informação. A Figura 34 mostra um exemplo de definição do tamanho de uma matriz em tempo de execução. Quando o algoritmo é executado, o usuário é requisitado a informar a



quantidade de pessoas (linhas). Nesse exemplo fixamos a quantidade de colunas em 5, as quais representarão as notas das atividades escolares, assim, saberemos o tamanho da matriz que precisamos criar. Por fim, esse tamanho é armazenado na variável “qtde”, a qual é utilizada pelo compilador *Java* na alocação de memória do *array* multidimensional.

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

System.out.println("Quantos alunos serão cadastrados?");
int qtde = dados.nextInt();

String notas [][] = new String[qtde][5];
```

Quantidade de linhas da matriz dependerá do valor informado pelo usuário. Já a quantidade de notas é fixa (5).

Figura 34 – Definindo o tamanho da matriz em tempo de execução.  
Fonte: próprio autor.

É importante ressaltar que a variável “qtde”, mostrada na Figura 34, pode não estar acessível em todo o programa, ou o conteúdo dessa variável pode ser alterado em algum outro ponto do algoritmo. Porém, essa informação é útil, por exemplo, para definir a condição de parada do laço de repetição. Então como descobrir posteriormente o tamanho da matriz? A resposta é simples. O *Java* disponibiliza uma propriedade em todas as matrizes chamada **length**. Se traduzirmos essa palavra para o português, ela quer dizer justamente, comprimento. Para buscar essa informação, basta usar o nome da matriz, o sinal de ponto final e a palavra reservada **length**, assim descobrimos a quantidade de linhas. Para descobrir a quantidade de colunas use essa mesma palavra, logo depois dos colchetes que indicam o índice das linhas.

A Figura 35 mostra um exemplo de uso. Vamos imaginar que não sabemos, de ante mão, qual o tamanho da matriz. Usamos então o comando “**notas.length**” para obter o tamanho de linhas da matriz. Assim, conseguimos criar uma condição de execução do laço. Ou seja, enquanto a variável de controle “i”, for menor que o tamanho de linhas da matriz, o laço de repetição continuará executando. Quando a variável de controle alcançar o comprimento, o laço é encerrado. O segundo laço usa o comando “**notas[i].length**” para obter a quantidade de colunas na linha apontada pelo índice “i”. Ainda nesse trecho de algoritmo, dentro do laço, imprimimos os dados de todas as células, às quais foram informadas pelo usuário.



```
double notas[][] = new double[10][5];

for (int i = 0; i < notas.length; i++){

    for (int j = 0; j < notas[i].length; j++){

        System.out.println(notas[i][j]);

    }

}
```

Retorna a quantidade de linhas da matriz.

Retorna a quantidade de colunas da linha atual da matriz.

Figura 35 – Exemplo de uso da propriedade “length”.

Fonte: próprio autor.



**Mídia digital:** Antes de prosseguirmos, assista ao vídeo “Tamanho da estrutura” para fazer uma revisão dessa seção.



**Dica do Professor:** É uma boa prática usar o *length* para se referenciar ao tamanho do vetor, mesmo quando conhecemos o tamanho do vetor, isso facilita futuras alterações no código.

## 2.6 Exemplos

Essa seção apresenta três exemplos de uso de matrizes. O objeto é reforçar o conteúdo aprendido até o momento. Para facilitar o entendimento, numeramos as linhas dos algoritmos e reforçamos que esses códigos foram digitados dentro de um método *main* de uma classe *Java* qualquer. A Figura 36 mostra o primeiro exemplo, a ideia aqui é criar um algoritmo que receba seis números em um formato de matriz, com duas linhas e três colunas e em seguida, imprima somente os valores ímpares. A linha 1 cria o objeto responsável pela leitura de dados. A linha 2 declara e instancia uma matriz do tipo *double* com 6 posições, ou seja, teremos os índices zero ao um para linhas e de zero a dois para as colunas. Para preenchermos toda a matriz usamos dois laços de repetição e fazemos a leitura dos dados (linhas 4 a 9). Nos dois últimos laços de repetição (linhas 10 a 14), percorremos todas as células e verificamos se o número é par ou ímpar. Para isso usamos o operador “%”. Ele é responsável por pegar o resto da divisão e essa divisão é de números inteiros, ou seja, não tem casa decimal. Se dividirmos o número por dois e o resto for diferente de zero, então o número é ímpar (linha 13). A linha 14 imprime o número se a condição for verdadeira.

```

1 Scanner dados = new Scanner(System.in);
2 double numeros[][] = new double[2][3];
3
4 for (int i = 0; i < numeros.length; i++){
5     for (int j = 0; j < numeros[i].length; j++){
6         System.out.printf("Digite o valor para a posição (%d,%d)", i, j);
7         numeros[i][j] = dados.nextInt();
8     }
9 }
10
11 for (int i = 0; i < numeros.length; i++){
12     for (int j = 0; j < numeros[i].length; j++){
13         if (numeros[i][j] % 2 != 0)
14             System.out.println(numeros[i][j]);
15     }
16 }

```

Figura 36 – Exemplo de algoritmo para impressão de números ímpares.

Fonte: próprio autor.

A Figura 37 apresenta um exemplo de algoritmo que armazena 20 números em uma matriz de cinco linhas e quatro colunas chamada “valores” e depois imprime qual é o menor valor armazenado. A linha 1 cria o objeto responsável pela leitura de dados. A linha 2 é responsável por declarar e instanciar uma matriz do tipo *int* com vinte posições (5 linhas com 4 colunas cada). Para preencher toda a matriz, foi usado dois laços de repetição, dentro deles requisitamos a entrada do usuário para todas as células (linhas 3 a 8). A grande novidade acontece da linha 10 a 15. Primeiro foi criada uma variável chamada “auxiliar”. Ela foi inicializada com o maior valor inteiro possível no *Java*. Isso vai garantir que qualquer número da matriz seja menor que esse valor inicial. Dentro do segundo laço (linha 13), verificamos se a posição atual, apontada pelas variáveis “i” e “j” têm um valor menor que o da variável chamada “auxiliar”, se for verdade, substituímos o valor atual da variável “auxiliar” pelo valor que está na célula atual. O algoritmo faz essa verificação para todas as posições. No final do laço, o menor valor é impresso na tela (linha 16).

```

1 Scanner dados = new Scanner(System.in);
2 int valores[][] = new int[5][4];
3 for (int i = 0; i < valores.length; i++){
4     for (int j = 0; j < valores[i].length; j++){
5         System.out.printf("Digite o valor para a posição (%d, %d)", i, j);
6         valores[i][j] = dados.nextInt();
7     }
8 }
9
10 int auxiliar = Integer.MAX_VALUE;
11 for (int i = 0; i < valores.length; i++){
12     for (int j = 0; j < valores[i].length; j++){
13         if (valores[i][j] < auxiliar)
14             auxiliar = valores[i][j];
15     }
16 System.out.println(auxiliar);

```

Figura 37 – Exemplo de algoritmo para descobrir o menor valor de uma matriz.

Fonte: próprio autor.

O terceiro e último exemplo de código fonte é exibido na Figura 38, esse algoritmo recebe 10 nomes e os armazenam em uma matriz de 2 linhas com 5 colunas cada e depois imprime esses dados em formato de tabela. A linha 1 cria o objeto responsável pela leitura de dados. A linha 2 declara e instancia uma matriz do tipo *String* com dez posições. Para preencher toda a matriz, foram usados dois laços de repetição, neles requisitamos a entrada do usuário para todas as células da estrutura (linhas 3 a 8). A lógica de imprimir os dados em formato de tabela acontece da linha 10 a 15. As linhas 12 e 14 são onde devemos analisar com mais detalhes. Repare que os dois laços percorrem todas as posições da matriz. Mas o comando de impressão na linha 12 não quebra a linha, depois de mostrar os dados da célula atualmente apontada pelas variáveis “i” e “j”, isso mantém todas as colunas de uma determinada linha na horizontal. Quando o laço mais interno percorre todas as colunas de uma linha, então forçamos a quebra de linha no *console* (linha 14). Isso imprime todas as células em formato de tabela.

```

1 Scanner dados = new Scanner(System.in);
2 String nomes[][] = new String[2][5];
3 for (int i = 0; i < nomes.length; i++){
4     for (int j = 0; j < nomes[i].length; j++){
5         System.out.printf("Digite o nome para a posição (%d, %d)", i, j);
6         nomes[i][j] = dados.next();
7     }
8 }
9
10 for (int i = 0; i < nomes.length; i++){
11     for (int j = 0; j < nomes[i].length; j++){
12         System.out.printf("%s\t\t", nomes[i][j]);
13     }
14     System.out.println("");
15 }

```

Figura 38 – Exemplo de algoritmo para imprimir os dados no formato da matriz.

Fonte: próprio autor.



**Mídia digital:** Volte à sala virtual e assista ao vídeo “Exemplos com matrizes”, a ideia é reforçar os conceitos apresentados até aqui nessa seção.



**Atenção:** Uma matriz deve ter duas ou mais dimensões e como os vetores, para cada dimensão os índices recebem o valor de zero até o seu tamanho menos um.



**Atenção:** O aprendizado de algoritmos requer muita prática, então refaça todos os exemplos apresentados até aqui, modifique-os e veja os resultados. Faça os exercícios propostos e procure material extra.

Concluimos assim a segunda semana de estudos, mais uma vez é hora de uma pausa para assimilar o conteúdo. Faça a leitura (ou releitura) de todo o texto, principalmente, assista aos vídeos propostos e analise todas essas informações.

Nos encontramos na próxima semana.

Bons estudos!



### Objetivos

Nessa terceira semana, nós vamos abordar alguns conceitos referentes a modularização dos algoritmos, a ideia é usar o conceito de métodos para criar códigos organizados e reaproveitáveis.

### 3.1 Considerações iniciais

Imagine que você está criando um programa com centenas de linhas de código, o qual apresenta diversas páginas/telas e, entre essas diversas porções de código, você precise repetir algoritmos para validar informações (datas, CPF, etc.), cálculos diversos (juros, quantidade de dias entre períodos) ou mostrar elementos na tela (cabeçalhos, campos de texto, etc.).

Uma solução seria copiar esses trechos de códigos e replicá-los nas diversas partes necessárias, ou seja, usar o famoso Ctrl+C Ctrl+V. Contudo, essa não é uma boa solução. Se for necessário modificar alguma instrução nesses trechos de código, o programador tem que localizar e modificar o algoritmo nos diversos fragmentos que compõe o programa e que foram copiados. O pior é que na correria do dia a dia, o programador pode esquecer de fazer as alterações em algumas dessas cópias. Isso geraria comportamentos diferentes e possíveis erros de execução do sistema. A solução para esse tipo de situação é usar métodos.

**Métodos** são trechos de código 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 chamados várias vezes durante a execução de um programa, ou seja, é uma sub-rotina que pode ser invocada sempre que necessário (Deitel, 2013). Entre as vantagens de uso desse recurso, pode-se citar:

- Dividir e estruturar um algoritmo em partes logicamente coerentes;
- Facilidade de testar os trechos em separado;
- Maior facilidade de manutenção do código;
- Evitar repetição do código-fonte (reduz o tamanho total de linhas de código);
- Maior legibilidade de um algoritmo.

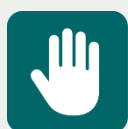
O termo “método” está vinculado aos conceitos do Paradigma Orientado a Objetos (o qual não é abordado no curso) e ele engloba duas definições importantes que são as **funções** e os **procedimentos**. Ambos os conceitos (funções e procedimentos) representam sub algoritmos, ou subprogramas e podem ou não receber valores de entrada, mas existe uma diferença importante entre eles, a qual é resumida na Tabela 1.

Método	Característica	Exemplo
Função	Sempre retorna um resultado a quem o chamou.	Calcular a multiplicação de dois números e retorna o resultado.
Procedimento	Não retornar um resultado a quem o chamou, ele apenas executa alguma ação.	Enviar documentos para a impressora.

Tabela 1 – Diferença entre função e procedimento.

Fonte: próprio autor.

Nas próximas seções vamos aprender como criar e executar os procedimentos e funções.



**Atenção:** Tenha em mente que um bom método deve ser autossuficiente, ou seja, possui internamente todas as definições (variáveis) necessárias para seu funcionamento. Evite também efetuar entrada/saída de dados dentro dos métodos assim ele ficará mais genérico.

## 3.2 Declaração

A declaração de um método fornece informações sobre ele, como visibilidade, tipo de retorno, nome e argumentos. A Figura 39 mostra com mais detalhes como é a sintaxe de definição desse tipo de porção de código.

```

public class Principal {

    //Nessa primeira linha tem-se o cabeçalho ou assinatura do método.
    qualificador [modificadores] tido-do-retorno nome-do-método([lista-de-argumentos]){
        // Corpo do método, o qual
        // pode conter ou não a palavra return
    }

    public static void main(String[] args) {

        System.out.println("Programa encerrado...");
    }
}

```

Declaração {

Início do método.

Fim do método.

Figura 39 – Sintaxe de definição de métodos em uma classe Java.

Fonte: próprio autor.

O primeiro ponto a ser entendido é que um método fica localizado dentro das classes *Java*, nesse caso, dentro da classe chamada “Principal”, mas eles ficam fora do método “main”. Uma classe pode ter quantos métodos forem necessários, mas sempre devem ser inseridos nesse local. Assim como o método “main”, os métodos devem ter uma chave de início e fim, essas chaves indicam o corpo do método e é nesse local aonde será implementado o algoritmo.

A primeira linha de definição do método é chamada de **assinatura** ou **cabeçalho**. É nesse ponto aonde definimos todo o comportamento do mesmo. Veja abaixo uma explicação mais detalhada de cada palavra do cabeçalho.

- **qualificador** – essa palavra define a visibilidade do método, para esse curso, vamos adotar sempre a palavra reservada **public**, o qual torna o método acessível a todas as classes do programa.
- **modificador** – são palavras reservadas que dão novos significados aos métodos, para esse curso vamos adotar sempre a palavra **static**, assim, o método “main”, o qual também é estático, poderá acessar o método sem problemas.
- **tipo-de-retorno** – aqui o programador deve informar o tipo de dado retornado pelo método. Métodos que não retornam valores devem conter a palavra reservada **void**, ou seja, nenhum valor será retornado após sua execução, é assim que informamos ao Java que esse método é um procedimento. Por outro lado, se quisermos retornar um valor, ou seja, criar uma função. Você pode inserir nesse ponto um tipo primitivo de dados (*int*, *float*, etc.), um vetor, uma matriz, ou ainda um objeto qualquer.
- **nome-do-método** – é um nome exclusivo usado para definir o nome de um método. Ele deve corresponder à funcionalidade do método, geralmente usa-se um verbo começando com letra em minúscula. O nome deve seguir as mesmas regras de nomeação de variáveis e classes.
- **lista-de-argumentos** – os parâmetros são inseridos dentro dos parênteses. Trata-se de uma lista de variáveis, as quais podem ser recebidas pelo método para 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. Os parâmetros são opcionais e como dito antes, podem ser uma lista, assim, usamos os colchetes para representar/ilustrar essa ideia, mas o “[” e “]” não são usados na definição real de um método. Por fim, é válido saber que mesmo se o método não tiver parâmetros, deixe os parênteses em branco, ou seja, sem nada dentro.

A melhor forma de entender todos esses conceitos é analisando alguns exemplos. A Figura 40 mostra um trecho de código com a criação de um procedimento chamado “imprimeCabeçalho”, esse método não tem parâmetros/valores de entrada. A cada vez que ele é chamado, ele imprime três linhas no *console* do computador. Repare que estamos usando o padrão para esse curso, ou seja, as palavras *public* e *static*.



Não retorna valor a quem chamar esse método. Ou seja, temos um procedimento.

```
public static void imprimeCabeçalho(){
    System.out.println("=====");
    System.out.println("-----IFMG-----");
    System.out.println("=====");
}
```

Corpo do método.

Figura 40 – Exemplo 01: criando um procedimento sem parâmetro de entrada.  
Fonte: próprio autor.

Agora, como um segundo exemplo, vamos criar um procedimento que recebe um texto e imprime esse texto no *console*. A Figura 41 mostra o método de nome “imprimeMensagem”, o qual implementa essa ideia. Veja que no cabeçalho do método tem-se um parâmetro de entrada, esse parâmetro de nome “texto” é usado dentro do comando de impressão.

O modificador **static** indica que o método não está vinculado a nenhum objeto (como dito, isso não será tratado nesse curso)

```
public static void imprimeMensagem(String texto){
    System.out.println(texto);
}
```

Existe um parâmetro de entrada no método.

O Corpo do método está usando o parâmetro de entrada, ou seja, o que estiver dentro da variável “texto” será impresso na tela.

Figura 41 – Exemplo 02: criando um procedimento com parâmetro de entrada.  
Fonte: próprio autor.

O terceiro exemplo dessa seção (Figura 42) mostra a implementação de uma função. Essa função se chama “somar”, ela recebe dois valores inteiros como parâmetro de entrada e retorna um valor do tipo inteiro com o resultado da soma. Repare no corpo do método que devemos usar a palavra **return** para indicar o que será retornado ao final da execução do método. Mais uma vez, começamos a implementação usando as palavras reservadas *public static*, depois informamos o tipo do retorno. É válido informar nesse ponto, que se o fluxo de execução do programa encontrar a palavra reservada **return**, o *Java* sai imediatamente do método e retorna o resultado a quem o chamou, isso acontece mesmo se existir linhas de código abaixo dessa palavra reservada.

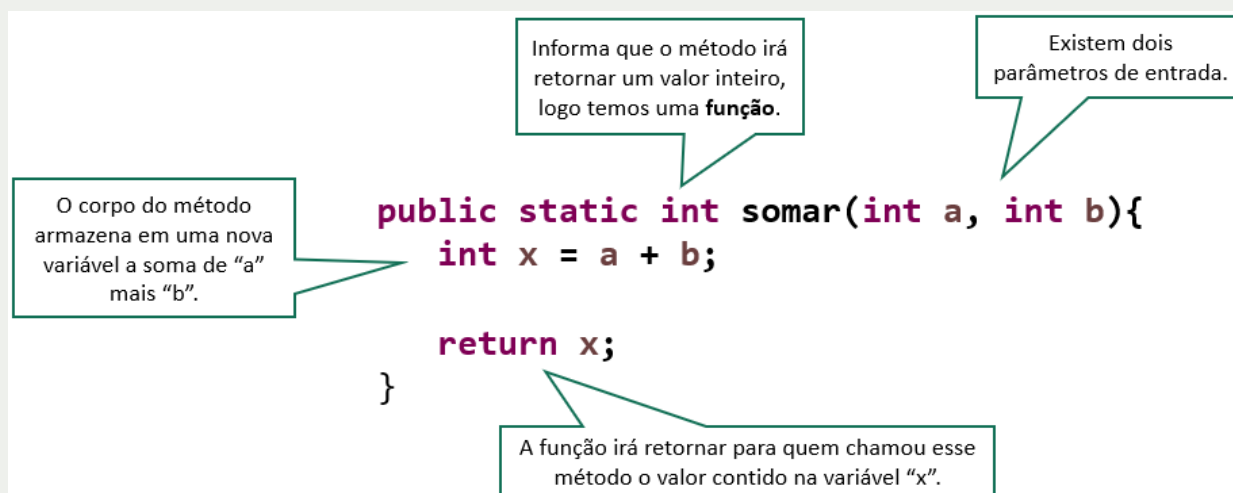


Figura 42 – Exemplo 03: criando uma função com parâmetro de entrada.  
Fonte: próprio autor.

A Figura 43 mostra o último exemplo dessa seção, aqui tem-se mais uma vez uma função. Mas agora ela recebe um vetor de inteiros e transforma esse vetor em um texto com todos os números separados por um caractere de espaço. Veja que no corpo do método tem um laço de repetição, o qual é responsável por acessar todas as posições do vetor e agrupar os números em uma *string* chamada "texto", ao final, o método retorna o conteúdo dessa variável.

```
public static String transformaVetorEmTexto(int[] vetor){

    String texto = "";
    for (int i=0; i< vetor.length; i++){
        texto = texto + vetor[i] + " ";
    }
    return texto;
}
```

Figura 43 – Exemplo 04: criando uma função com um veto como parâmetro de entrada.  
Fonte: próprio autor.



**Mídia digital:** Volte à sala virtual e assista ao vídeo “Métodos – Introdução”, a ideia é reforçar os conceitos apresentados até aqui.



**Mídia digital:** Volte mais uma vez à sala virtual e assista ao vídeo “Métodos – Retorno de valores”, assim, você está pronto para continuar nesse capítulo.

### 3.3 Ativação e fluxo de execução

A definição de um método não implica que ele será executado. Para o método ser executado, o programador precisa chamar ou ativar esse método no *main*. A chamada é feita pelo nome dele seguido de seus parâmetros de entrada, caso ele assim o exija. A Figura 44 mostra a chamada dos três primeiros métodos criados na seção anterior. Primeiro, o código chama o procedimento que imprime o cabeçalho no *console*. Logo em seguida, o método que imprime uma mensagem no console é executado. Repare que o primeiro método não tem parâmetros, então usamos apenas o abre e fecha parênteses. Já na segunda chamada passamos um parâmetro do tipo *String*, justamente o que a definição do método exige. O que estiver dentro dos parênteses será o texto impresso no *console*.

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

    imprimeCabeçalho();

    imprimeMensagem("Vamos somar 10 com 15");

    int x=10;
    int y=15;

    int t = somar(x, y);

    imprimeMensagem("O resultado é "+t);

}
```

Figura 44 – Exemplo de chamadas de métodos.  
Fonte: próprio autor.

Um exemplo bem interessante é a chamada da função responsável por somar dois números (Figura 44). Os parâmetros de entrada dessa função foram passados via variáveis, ou seja, os valores somados serão os que estão dentro das variáveis “x” e “y”. Como dito na Seção 3.1, uma função retorna um valor, assim foi criada a variável “t” para armazenar o resultado produzido pela função “somar”. Por fim, a função “imprimeMensagem” foi utilizada novamente, agora para imprimir o resultado da soma dos dois números. Aqui podemos ver na prática o reaproveitamento de código, usamos duas vezes a função de imprimir mensagens no console. Se por acaso, o programador mudar a implementação dessa função, a mudança será refletida nesses dois pontos.

Para entender um pouco mais da ativação do primeiro procedimento, a Figura 45 exhibe setas mostrando como é o fluxo de execução. Tudo começa pela seta “Início”.

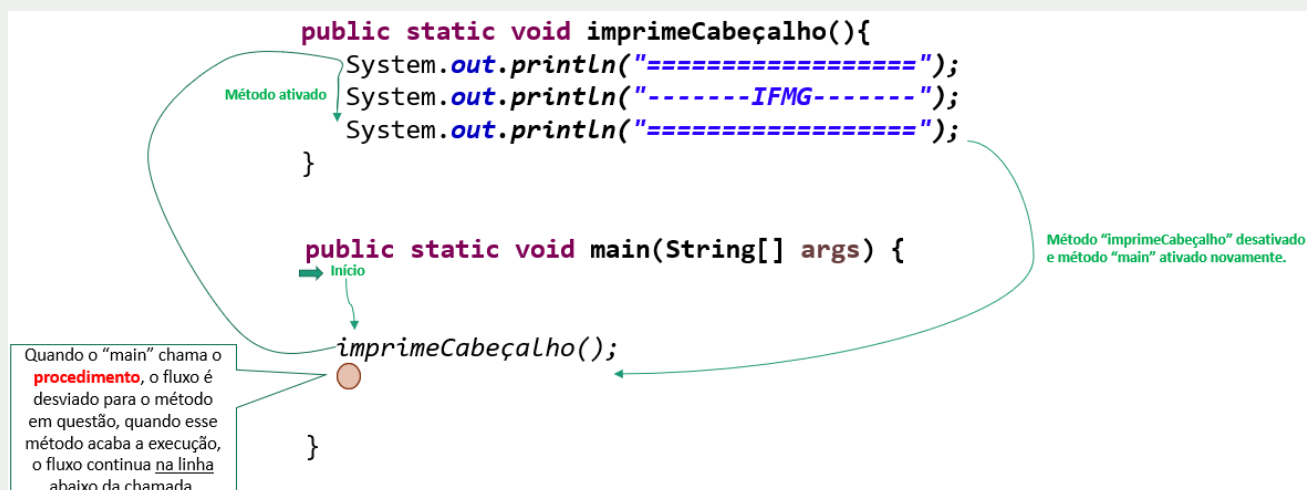


Figura 45 – Exemplo de fluxo de execução de um procedimento.

Fonte: próprio autor.

O ponto de retorno do procedimento é na linha logo abaixo da chamada do método. Já em uma função, o ponto de retorno é logo à frente da chamada, ou seja, atribuindo o retorno à uma variável. Veja a Figura 46, mais uma vez, siga as setas para acompanhar o fluxo de execução. Nessas duas figuras que mostram o fluxo de execução, têm um círculo indicando o ponto de retorno dos métodos.

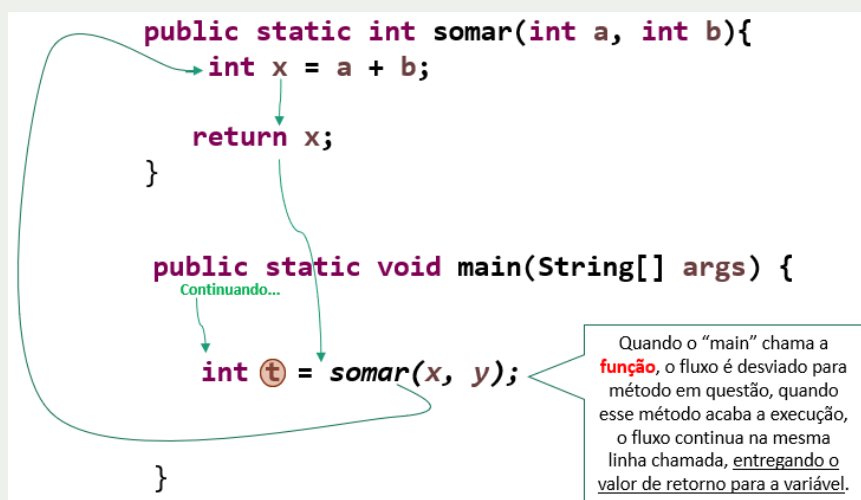


Figura 45 – Exemplo de fluxo de execução de uma função.

Fonte: próprio autor.

### 3.4 Variáveis locais

Dentro dos métodos, todas as regras de sintaxes e semântica são idênticas ao que é aplicado no *main*. Assim, o programador pode fazer uso de instruções sequencias, laços de repetição, definição de variáveis, objetos, vetores e matrizes. Contudo, a declaração ou instanciação dessas posições de memória tem um escopo local, ou seja, qualquer variável ou objeto criado em um método só é acessível dentro do método.

Como as variáveis internas tem escopo local, o programador pode inclusive usar nomes que já existem em outros métodos. Outro ponto interessante é que, as variáveis declaradas internamente são criadas na memória, assim que o método é ativado e o fluxo de execução atinge a linha da declaração, por fim, todas as variáveis declaradas são destruídas quando o método acaba a execução, o interessante é que o próprio *Java* faz esse controle da memória para o programador. A Figura 46 mostra exemplos hipotéticos de declarações de variáveis em três métodos diferentes. As setas em verde explicitam as declarações das variáveis internas.

```
public void imprimeVariasLinha (int qtdeLinha)
{
➡ int i = 1;
  while (i <= qtdeLinha)
  {
    System.out.println("=====");
    i += 1;
  }
}

public String nomeDoAluno ()
{
➡ String aluno = "Tião José Silva";
  return aluno;
}

public float areaDoQuadrado(int base, int altura)
{
➡ float area = 0;
  area = base * altura;
  return area;
}
```

Figura 46 – Exemplo de declaração de variáveis locais aos métodos.  
Fonte: próprio autor.

### 3.5 Parâmetros de entrada

Já aprendemos que os métodos em *Java* podem receber valores, os quais são passados via parâmetros de entrada. Mas falta aprender que existem dois tipos de passagem de parâmetro. Na primeira forma, é passada uma cópia do valor da variável no parâmetro de entrada. No segundo tipo, é passada uma referência à própria variável, ou seja, a própria posição de memória dela.

O primeiro método é chamado de **passagem de parâmetro por valor**. Quando o conteúdo do parâmetro passado dessa forma for alterado dentro do método, essas alterações não são refletidas fora dele. Em *Java*, todos os tipos primitivos de dados são passados por valor (*int*, *float*, *boolean*, *double* e *char*). Para reforçar esse conceito, vamos analisar as Figuras 47 e 48, as quais mostram respectivamente o código fonte e o

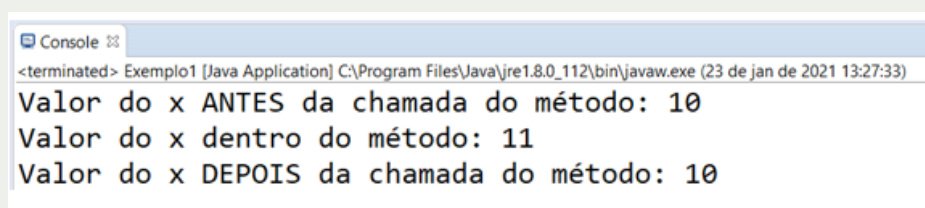
resultado da execução do mesmo. No código fonte, tem-se uma função que incrementa uma variável “x” em um, imprime o resultado dentro do método e retorna o novo valor. No método main, temos também uma variável chamada “x”, a qual é iniciada com o valor dez. Essa variável é passada como parâmetro. Repare que imprimimos o valor de “x” antes e depois de chamar o método para incrementar o valor em um.

```
public static int incrementa(int x){
    x = x + 1;
    System.out.println("Valor do x dentro do método: " + x);
    return x;
}

public static void main(String[] args) {
    int x = 10;
    System.out.println("Valor do x ANTES da chamada do método: " + x);
    incrementa(x);
    System.out.println("Valor do x DEPOIS da chamada do método: " + x);
}
```

Figura 47 – Exemplo de código para explorar a passagem de parâmetro por valor.  
Fonte: próprio autor.

Ao visualizar o resultado de execução do algoritmo na Figura 48, perceba que foram impressos os valores 10, 11 e 10. Isso aconteceu porque a variável “x”, foi modificada dentro do método, mas essa modificação não se reflete de fora dele, esse é justamente o comportamento da passagem de parâmetro por valor. Assim, foi impresso o valor inicial (10), depois o valor dentro do método (11), e por fim, o valor final de “x” que não foi afetado pela função, por isso continua com o valor 10. Nesse exemplo, o retorno da função foi desconsiderado.



```
<terminated> Exemplo1 [Java Application] C:\Program Files\Java\jre1.8.0_112\bin\javaw.exe (23 de jan de 2021 13:27:33)
Valor do x ANTES da chamada do método: 10
Valor do x dentro do método: 11
Valor do x DEPOIS da chamada do método: 10
```

Figura 48 – Resultado da execução do algoritmo que usa a passagem de parâmetro por valor.  
Fonte: próprio autor.

Já no método de **passagem de parâmetro por referência**, quando o conteúdo passado dessa forma for alterado dentro do método, essas alterações são refletidas fora dele. Em *Java*, todos os objetos e vetores são passados dessa forma, ou seja, todos os elementos que você usa a palavra “new” para instanciá-los.

Mais uma vez, para reforçar esse conceito, vamos analisar as Figuras 49 e 50, as quais mostram respectivamente o código fonte e o resultado da execução do mesmo. No código fonte, tem-se uma função que retorna uma representação em *String* do vetor e uma

outra função chamada “inicializaVetor”, a qual realmente modifica o vetor colocando o valor zero em todas as posições. A ideia aqui é parecida com o exemplo anterior, vamos imprimir os dados antes e depois de chamar um método que modifica o parâmetro.

```
public static String transformaVetorEmTexto(int[] vetor){
    String texto = "";
    for (int i=0; i< vetor.length; i++)
        texto = texto + vetor[i] + " ";
    return texto;
}

public static void inicializaVetor(int[] vetor){
    for (int i=0; i< vetor.length; i++)
        vetor[i] = 0;
    System.out.println( transformaVetorEmTexto(vetor) );
}

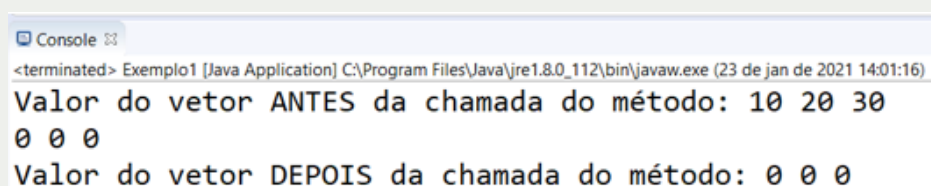
public static void main(String[] args) {

    int x[] = {10, 20, 30};
    System.out.println("Vetor ANTES da chamada do método: " + transformaVetorEmTexto(x));
    inicializaVetor(x);
    System.out.println("Vetor DEPOIS da chamada do método: " + transformaVetorEmTexto(x));
}
```

Figura 49 – Exemplo de código para explorar a passagem de parâmetro por referência.

Fonte: próprio autor.

Ao visualizar o resultado de execução do algoritmo da Figura 49, perceba que foram impressos os valores originais, depois o vetor com todas as posições com o valor zero por duas vezes. Isso aconteceu porque o vetor “x”, foi modificado dentro do método e essa modificação é refletida de fora dele, esse é justamente o comportamento da passagem de parâmetro por referência. Então, primeiro foi impresso os valores originais “10 20 30”, depois o método “inicializaVetor” colocou o valor zero em todas as posições e imprimiu os dados do vetor “0 0 0”. Por fim, já no método *main*, foi impresso o vetor que foi modificado dentro procedimento “inicializaVetor”, assim os valores “0 0 0” foram mantidos.



```
Console
<terminated> Exemplo1 [Java Application] C:\Program Files\Java\jre1.8.0_112\bin\javaw.exe (23 de jan de 2021 14:01:16)
Valor do vetor ANTES da chamada do método: 10 20 30
0 0 0
Valor do vetor DEPOIS da chamada do método: 0 0 0
```

Figura 50 – Resultado da execução do algoritmo que usa a passagem de parâmetro por referência.

Fonte: próprio autor.





**Mídia digital:** Volte à sala virtual e assista ao vídeo “Métodos – Tipos de passagem de parâmetros”, a ideia é reforçar os conceitos apresentados e ver mais alguns exemplos.

### 3.6 Exemplos

Para finalizarmos essa terceira semana, nós vamos analisar três exemplo de uso de métodos. O primeiro exemplo mostra a criação e a chamada de uma função que calcula o valor de um produto qualquer, isso de acordo com um percentual de desconto. Assim, esse método recebe dois valores do tipo *float* como parâmetro de entrada, os quais representam o valor do produto e a porcentagem de desconto (linha 1), ao final ele retorna o valor do produto menos o desconto. Como se trata de uma função, temos que usar a palavra reservada *return* (linha 2) pra indicar o que será retornado. A linha 9, mostra a chamada do método dentro do main. Repare que o valor retornado pela função já vai ser inserido/utilizado pelo “System.out.println” para ser impresso na tela.

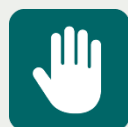
```

1 public static float calculaDesconto(float valor, float desconto){
2     return valor - (valor * desconto/100);
3 }
4
5
6
7 public static void main(String[] args) {
8
9     System.out.println(calculaDesconto(1000.0f, 5f));
10 }

```

Figura 51 – Exemplo de função para cálculo de desconto.

Fonte: próprio autor.



**Atenção:** Durante a execução da função, se a palavra reservada “*return*” for encontrada/atingida, então implica na saída imediata do método. Assim, a execução continua no ponto em que tal método foi chamado.

O segundo exemplo (Figura 52) mostra o código de um procedimento, o qual simplesmente imprime na tela as opções de um menu hipotético. Sabe-se que esse método não retorna valor por causa da palavra reserva “*void*” na linha 1. No método main temos apenas uma chamada a esse método (linha 12). Aqui tem-se um exemplo de como o código fica mais limpo com o uso de procedimentos/funções. Se fosse preciso chamar o menu do programa, bastaria usar o comando “menu();”, ao invés, de digitar as seis linhas que apresentam as opções.



```

1 public static void menu(){
2     System.out.println("-----Menu-----");
3     System.out.println("1) Inserir produto");
4     System.out.println("2) Excluir produto");
5     System.out.println("3) Listar produtos");
6     System.out.println("4) Sair do programa");
7     System.out.println("Escolha uma opção");
8 }
9
10 public static void main(String[] args) {
11
12     menu();
13 }

```

Figura 52 – Exemplo de procedimento para impressão de um menu na tela.  
Fonte: próprio autor.

Para finalizar os exemplos, a Figura 53, apresenta uma função que é responsável por inverter os dados do vetor passado como parâmetro. Como o vetor é passado por referência, não precisamos usar uma função, pois já estamos alterando o vetor original. As linhas 13 e 14, respectivamente, declaram/instanciam um vetor e chamam o procedimento, que foi definido entre as linhas 1 a 9. A lógica de inversão é simples. Mantemos dois índices, um que começa da posição zero e outro que começa na última posição do vetor, fazemos a troca dos dados usando uma variável auxiliar (linha 5), por fim, incrementamos o índice “i” e decrementamos o índice chamado “limite”. Fazemos esse processo até a metade do vetor, assim conseguimos inverter todos os dados.

```

1 public static void inverterVetor(int[] vetor){
2
3     int limite = vetor.length -1;
4     for (int i = 0; i < (vetor.length/2); i++) {
5         int aux = vetor[i];
6         vetor[i] = vetor[limite-i];
7         vetor[limite-i] = aux;
8     }
9 }
10
11 public static void main(String[] args) {
12
13     int[] numeros = {3,5,7,9,100,12};
14     inverterVetor(numeros);
15 }

```

Como o vetor é passado por referência, o que for alterado dentro do método será alterado de fora também.

Figura 53 – Exemplo de função que inverte as posições dos dados de um vetor.  
Fonte: próprio autor.



**Atenção:** Sempre que possível, tente implementar seus algoritmos com o uso de métodos, assim, seu código tende a ser mantido e reaproveitado mais facilmente.

Concluimos nossa penúltima semana de estudos. Assista aos vídeos propostos e analise todas essas informações com cuidado e quantas vezes forem necessárias.

Nos encontramos na próxima semana.

Bons estudos!



### Objetivos

Nessa última semana, você aprenderá sobre a manipulação de arquivos dos tipos binário e texto.

### 4.1 Conceitos iniciais

Até o momento, falamos muito sobre variáveis, vetores e matrizes, os quais servem para armazenar conteúdos em memória. Mas é fácil perceber que as informações registradas nesses tipos de estruturas de dados são perdidas a cada execução dos algoritmos. Assim, algumas perguntas surgem: Por que isso acontece? Como manter informações mesmo que o computador seja reiniciado (desligado e ligado novamente)?

Primeiro é importante saber que os dados armazenados nas variáveis, vetores e matrizes ficam na memória principal do computador (Memória RAM), a qual também é chamada de memória **volátil**. Esse tipo de memória é muito rápida e conversa diretamente com o processador, mas ela “paga caro” por isso, seus componentes eletrônicos não conseguem guardar os dados de forma permanente. Como os tipos primitivos de dados em *Java* trabalham sobre essa memória, os dados são perdidos a cada execução.

Para persistir os dados de forma não volátil é preciso utilizar recursos que permitam a gravação de informações na **memória secundária** (HDs, Pendrives, discos, etc.), esses são os tipos de memória que conseguem persistir os dados de forma permanente. As estruturas de dados armazenadas na memória secundária recebem o nome de **arquivos** (ASCENIO, 2008). Para Deitel (2013), um arquivo é uma abstração utilizada para uniformizar a interação entre o ambiente de execução (um programa) e os dispositivos secundários (o disco rígido, por exemplo).

*Java* permite trabalhar com arquivos sem a preocupação com o sistema operacional, isso torna o trabalho mais fácil e de alto nível. Além disso, pode-se gravar e/ou ler bytes (arquivos binários), ou caracteres (arquivos de texto).

No primeiro, tem-se uma sequência de *bytes*. Um byte é composto de oito *bits*, ou seja, oito unidades que podem assumir os valores 0 ou 1. Por exemplo, o valor 9 em bytes teria a forma 1001. Esses arquivos armazenam os dados de forma literal, ou seja, não são caracteres. Geralmente, esse tipo de arquivo não tem o conceito de quebra de linha. Pode-se imaginá-los como uma fita sequencial cheia de dados, ou seja, *bytes*.

Já nos fluxos de entrada e saída em arquivos de texto, armazenam-se e recuperam-se dados como uma sequência de caracteres dividida em linhas, as quais são terminadas por um caractere especial de final de linha (“\n”).

Existem diversas classes em *Java* para manipulação dos arquivos e do conteúdo deles, todas pertencentes ao pacote **java.io**. Esse pacote divide os códigos fontes entre classes que seguem uma hierarquia, nesse ponto, vamos abstrair que uma **classe** é um

“miniprograma” que implementa um ou mais algoritmos responsáveis por resolver o problema que ele se propõe. A Figura 54 mostra parte da hierarquia de classes para manipulação de arquivos. Não se preocupe em decorar nomes. Atente-se em entender como essas classes estão relacionadas e como elas funcionam. Sempre que precisar, consulte a documentação ou esse documento para buscar exemplos e modificá-los

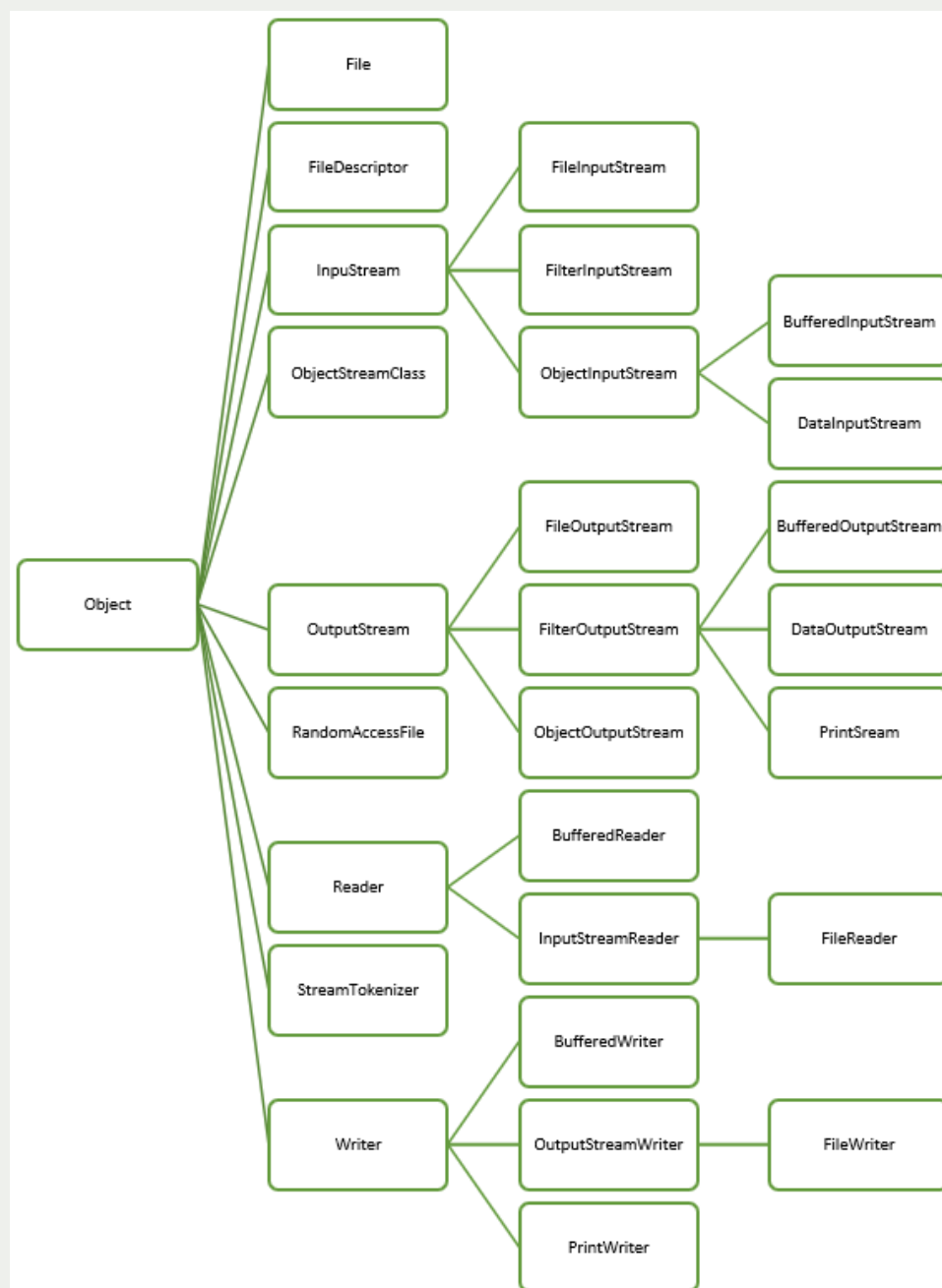


Figura 54 – Parte da hierarquia de classes para manipulação de arquivos em Java.  
Fonte: Kholedov Igor – Bistrol Community College<sup>1</sup>.

Quando trabalhamos com arquivos, diversos eventos externos podem acontecer, por exemplo, o arquivo pode não estar no local informado ou, seu usuário não tem

<sup>1</sup> [http://www.c-jump.com/bcc/c257c/Week11/W11\\_0010\\_io\\_class\\_hierarchy.htm](http://www.c-jump.com/bcc/c257c/Week11/W11_0010_io_class_hierarchy.htm) Acessado em 07/06/2022.

permissão de acesso ao conteúdo salvo. Assim, as classes do pacote **java.io** nos obrigam a tratar **exceções**. Como esse assunto não é o escopo desse curso, vamos adotar a forma mais simples de tratamento de erros/exceções, iremos adotar a estratégia de incluir uma cláusula *thrown* no cabeçalho do método *main* da classe. Veja na Figura 55 do que se trata essa estratégia. Para todos os exemplos deste capítulo, nós usaremos o tratamento de exceção mais genérico possível, o que é feito pela interface *IOException*. Nessa abordagem, em qualquer erro externo ao algoritmo, o programa irá simplesmente encerrar a execução. Nós também faremos a importação de todas as classes de manipulação de arquivos do *Java*, isso vai facilitar a codificação e diminuir o tamanho do algoritmo.

```
import java.io.*;

public class Principal {

    public static void main(String[] args) throws IOException {

        //Vamos inserir nossos códigos sobre arquivos aqui...

    }

}
```

Para facilitar, vamos importar todas as classes do pacote responsável pela entrada e saída de dados.

Para facilitar, não vamos fazer um tratamento superficial de erros, vamos apenas informar ao Java que em qualquer problema, ele pode encerrar o programa.

Figura 55 – Padrão de código em uma classe *main* para manipulação de arquivos em *Java*.  
Fonte: próprio autor.



**Mídia digital:** Volte à sala virtual e assista ao vídeo “Introdução à manipulação de arquivos”, a ideia é reforçar os conceitos iniciais desse assunto tão importante.

## 4.2 A classe File

A primeira classe que vamos trabalhar se chama **File**. Ela não consegue ler ou gravar conteúdo em um arquivo. Porém, ela fornece meios de associar uma variável a um arquivo físico, assim podemos manipular o arquivo físico no disco, ou seja, conseguiremos por exemplo, criar o apagar um arquivo em um *Pendrive*, via código fonte. Na Figura 56 podemos ver duas opções de criação de um objeto da classe *File*, o qual foi nomeado de “arq”. Agora essa variável pode referenciar um arquivo externo ao nosso programa. Entre aspas temos um caminho (path) do arquivo ou diretório. Esse caminho pode não existir, por isso falamos que ele é abstrato. O caminho do arquivo deve ser uma *string* ou uma

variável desse tipo. Nessa figura estamos referenciando um arquivo chamado “cadastro” e ele tem a extensão “txt”.

O diagrama apresenta dois exemplos de criação de objetos da classe `File` em Java, com explicações para cada um.

**Exemplo 1:**

```
File arq = new File("cadastro.txt");
```

ou

**Exemplo 2:**

```
File arq = new File("c:\\curso\\testes\\cadastro.txt");
```

**Explicação para o Exemplo 1:** Nesse caso, o Java procura o arquivo “cadastro.txt” no próprio diretório do seu algoritmo/programa.

**Explicação para o Exemplo 2:** Aqui é importante lembrar que a “\” é um caractere especial dentro de uma string, pois ela pode ser usada para quebrar uma linha “\n” criar uma tabulação “\t”, entre outros. Assim, usamos “\\” para indicar que não estamos usando o caractere especial e, literalmente, a barra “\”. Nesse caso, o Java procura o arquivo dentro do diretório chamado “teste” que está dentro de outra pasta chamada “curso”, a qual está no driver “C” do computador.

Figura 56 – Exemplo de criação de um objeto do tipo `File`.

Fonte: próprio autor.

A Figura 57 mostra um algoritmo com o primeiro exemplo de uso da classe `File`. Na linha 1, criamos a variável chamada “arq”, o qual representa o arquivo externo. Veja que o caminho aponta para o drive “c” do computador e nesse drive existe uma hierarquia de diretórios, ou seja, o arquivo chamado “cadastro.txt” está dentro de uma pasta chamada “teste”, a qual está dentro da pasta “curso”. Feita essa vinculação entre arquivo e variável, o algoritmo apresenta dois condicionais (linhas 3 e 9).

No primeiro condicional foi utilizado o método “`.exists()`”. Ele é utilizado para verificar se esse arquivo existe no caminho informado ou não. Se o arquivo “casdastro.txt” existir nesse local, o método retorna *true*, caso contrário *false*. Então usamos esses valores para imprimir diferentes mensagens na tela.

No segundo condicional (*if*) utilizamos o método “`.canRead()`”. Esse método é responsável por analisar se o programa, ou usuário que executou o programa têm a permissão de leitura sobre esse arquivo. Ele verifica também se outra pessoa está editando/bloqueando o arquivo no momento da execução do algoritmo. Assim, o método retornar verdadeiro se é possível ler o arquivo, caso contrário ele retorna falso. Mais uma vez, para esse exemplo mostramos mensagens diferentes, de acordo com o retorno do método. Similar a esse método, existe o “`.canWrite()`”. Mas ele é responsável por analisar se o programa, ou usuário que executou o programa, têm a permissão de escrita sobre esse arquivo.

```

1 File arq = new File("c:\\curso\\testes\\cadastro.txt");
2
3 if (arq.exists()) {
4     System.out.println("Arquivo/diretório existe!");
5 } else {
6     System.out.println("Arquivo/diretório NÃO existe!");
7 }
8
9 if (arq.canRead())
10     System.out.println("O arquivo pode ser lido.");
11 else {
12     System.out.println("O arquivo NÃO pode ser lido.");
13     System.exit(-1);
14 }

```

Figura 57 – Primeiro exemplo de uso de um objeto do tipo File.

Fonte: próprio autor.

A Figura 58 mostra mais dois métodos novos da classe *File*. Na linha 4, se o conteúdo apontado pela variável “arq” for um diretório, o programa imprime a mensagem “Esse é um diretório”, na linha 7 utilizamos o método “.isFile()” para descobrir se estamos lidando com um arquivo.

```

1 File arq = new File("c:\\curso\\testes\\cadastro.txt");
2
3 if (arq.exists() && arq.canRead()){
4     if (arq.isDirectory())
5         System.out.println("Esse é um diretório.");
6
7     if (arq.isFile())
8         System.out.println("Esse é um arquivo.");
9
10 } else {
11     System.out.println("Operação não permitida.");
12 }

```

Figura 58 – Segundo exemplo de uso de um objeto do tipo File.

Fonte: próprio autor.

A Figura 59 mostra o terceiro e último exemplo de uso da classe *File*. Na linha 1 foi definida uma variável chamada “dir”, repare que na *string* que representa o caminho (path), não existe uma extensão, assim, estamos nos referindo a um diretório, nesse caso, um diretório chamado “novo”. Na linha 2 é feito um condicional para verificar se o diretório existe, nesse caso estamos usando o operador de negação (!), a ideia é criar a condição:



“se NÃO existir o diretório”. Se essa condição for verdadeira, o algoritmo executa o método “*mkdir()*”, o qual é responsável por criar o diretório chamado “novo”.

Ainda na Figura 59. Criamos outro objeto do tipo *File* chamado “*arq*” (linha 11). Agora estamos apontando para um arquivo, pois o caminho (*path*) termina com a extensão “.txt”. Com a variável “*arq*”, o algoritmo apresenta mais dois novos métodos. O primeiro (linha 12), chamado “*length()*”, retoma o tamanho do arquivo ou diretório em *bytes*. Já na linha 15, o algoritmo chama o método “*delete()*” para apagar o arquivo “cadastro.txt” da memória secundária. Se a execução do método for bem sucedida, o retorno é *true*, caso contrário *false*. Assim, podemos montar um condicional, a ideia é dar um *feedback* amigável ao usuário (linhas 15 a 18).

```
1 File dir = new File("c:\\curso\\testes\\novo");
2 if (!dir.exists()){
3
4 if (dir.mkdir())
5     System.out.println("Diretório criado com sucesso.");
6 else
7     System.out.println("Erro na criação do diretório.");
8
9 }
10
11 File arq = new File("c:\\curso\\testes\\cadastro.txt");
12 long tamanho = arq.length();
13 System.out.println("Tamanho do arquivo: " + tamanho);
14
15 if (arq.delete())
16     System.out.println("Exclusão realizada com sucesso.");
17 else
18     System.out.println("Erro durante a exclusão.");
```

Figura 59 – Terceiro exemplo de uso de um objeto do tipo *File*.

Fonte: próprio autor.



**Mídia digital:** Volte à sala virtual e assista ao vídeo “Introdução à classe *File*”, a ideia é reforçar os conceitos apresentados e ver a execução dos algoritmos de manipulação de arquivo na prática.

### 4.3 Gravando dados em arquivos binários

O próximo passo no aprendizado de manipulação de arquivos é saber com o conteúdo do arquivo pode ser salvo. Essa seção será dedicada a gravação dos dados do tipo binário. Muitos tipos de arquivos trabalham dessa forma, por exemplo, um arquivo de música ou vídeo. Nesse formato, os dados não ficam legíveis para nós humanos, mesmo se gravarmos um texto. Para manipular arquivos binários podemos usar duas classes:

- **FileOutputStream** – é uma classe que grava dados binários (bytes) no arquivo. Você deve informar para essa classe um objeto do tipo `File` ou um nome do arquivo em string.
- **DataOutputStream** – Se tentarmos gravar dados que estão em variáveis (tipos primitivos de dados) em um arquivo binário, nós teríamos um problema, pois são formatos diferentes. Mas com essa classe conseguimos converter os dados dos tipos primitivos para bytes. Lembre-se, se o dado já estiver no formato binário, não precisaríamos desse objeto. Um objeto dessa classe recebe um `FileOutputStream` para gravar os dados.

A Figura 60 mostra um exemplo de uso das duas classes citadas acima. Como dito, elas trabalham em conjunto. Veja que nas linhas 1 e 2, um objeto do tipo `FileOutputStream` com o nome de “arquivo” é criado. Ele faz referência ao arquivo físico na memória secundária com o nome de “cadastro.dat”. A linha 3 cria a variável chamada “gravarArquivo” e esse passo só é possível, pois o `DataOutputStream` recebeu como parâmetro, justamente, o objeto criado nas primeiras duas linhas.

As linhas 5 até 8 mostram os comandos para efetivamente gravar conteúdo no arquivo. Veja que existe um método para cada tipo de dados. Por exemplo, o método “`writeChar`” escreve um caractere, o método “`writeDouble`” escreve um valor decimal. Esse trecho de código da Figura 60 não mostrou, mas as palavras nome, sexo, idade e salário são variáveis dos respectivos tipos primitivos: `String`, `char`, `int` e `double`. Por fim, é interessante notar que a linha 10 usa o método “`.close()`”. Esse passo é importante porque estamos liberando o arquivo para ser usado por outras pessoas ou programas.

```
1 FileOutputStream arquivo =  
2     new FileOutputStream("c:\\curso\\testes\\cadastro.dat");  
3 DataOutputStream gravarArquivo = new DataOutputStream(arquivo);  
4  
5 gravarArquivo.writeUTF(nome);  
6 gravarArquivo.writeChar(sexo);  
7 gravarArquivo.writeInt(idade);  
8 gravarArquivo.writeDouble(salario);  
9  
10 gravarArquivo.close();
```

Figura 60 – Uso de classes Java para gravação de dados binários.  
Fonte: próprio autor.

## 4.4 Lendo dados de arquivos binários

Agora vamos fazer o caminho contrário, ou seja, ler o conteúdo do arquivo binário. Repare que a ideia é similar à gravação. Para leitura de arquivos nesse formato podemos usar essas duas classes:

- **FileInputStream** – é uma classe que lê dados binários (bytes) no arquivo. Você deve informar para essa classe um objeto do tipo *File* ou um local e nome do arquivo em string.
- **DataInputStream** – Lembre-se, os dados estão no formato binário. Assim, usamos esse objeto para converter esses dados, isso deve ser feito antes de atribuir os valores para as variáveis. Assim, um objeto dessa classe recebe um objeto *FileInputStream* como parâmetro de entrada.

A Figura 61 mostra um exemplo de uso das classes citadas acima. Elas também trabalham em conjunto. Veja que nas linhas 1 e 2, um objeto do tipo *FileInputStream* com o nome de “arquivo” é criado. Ele faz referência ao arquivo físico na memória secundária, o qual tem o nome de “cadastro.dat”. A linha 3 cria a variável chamada “lerArquivo” e esse passo só é possível, pois o *DataInputStream* recebeu como parâmetro, justamente, o objeto criado nas duas primeiras linhas.

As linhas 5 até 8 mostram os comandos, para efetivamente, buscar os conteúdos no arquivo. Veja que existe um método para cada tipo de dados. Por exemplo, o método “readChar” lê um caractere, o método “readDouble” lê um valor decimal. Para cada um desses métodos de leitura foi declarada uma variável, elas recebem os dados convertidos. Logo em seguida, as variáveis são impressas na tela (linhas 10 até 13). Por fim, na linha 15 usamos o método “.close()” para fechar o arquivo. Esse passo é importante para desbloquear o arquivo.

```
1 FileInputStream arquivo =
2     new FileInputStream("c:\\curso\\testes\\cadastro.dat");
3 DataInputStream lerArquivo = new DataInputStream(arquivo);
4
5 String nome = lerArquivo.readUTF();
6 char sexo = lerArquivo.readChar();
7 int idade = lerArquivo.readInt();
8 double salario = lerArquivo.readDouble();
9
10 System.out.println("Nome: "+nome);
11 System.out.println("Sexo (M/F): "+sexo);
12 System.out.println("Idade: "+idade);
13 System.out.println("Salário: "+salario);
14
15 lerArquivo.close();
```

Figura 61 – Uso de classes Java para leitura de dados binários.  
Fonte: próprio autor.



**Mídia digital:** Volte à sala virtual e assista ao vídeo “Manipulação de arquivos binários”, o objetivo é reforçar o uso das classes de forma prática.

## 4.5 Lendo dados de arquivos texto

Devido à sua simplicidade, arquivos texto são comumente utilizados para armazenamento de informações. Além disso, o formato de dados são legíveis a nós, humanos. Contudo, mesmo trabalhando com texto, o Java usa uma classe que manipula bytes chamada **FileInputStream** para, efetivamente, ler os dados no disco:

```
FileInputStream arquivo = new FileInputStream("arquivo.txt");  
int b = arquivo.read();
```

O problema é que o método “read”, nos retorna apenas um byte, ou um caractere em representação numérica. Assim, teremos que usar classes intermediárias afim de fazer a tradução para o formato de letras que conhecemos. Contudo, para traduzir o *byte* para caractere usa-se a classe **InputStreamReader**. Mas para criar um objeto dessa classe informamos o **FileInputStream** criado no passo anterior:

```
InputStreamReader conversor = new InputStreamReader(arquivo)
```

O problema é que a classe acima não consegue pegar uma quantidade maior de dados, isso é feito pela **BufferedReader**:

```
BufferedReader dados = new BufferedReader(conversor);  
String s = dados.readLine();  
System.out.println(s);  
dados.close();
```

A Figura 62 resume todo o processo de composição de classes *Java* para manipular os dados no formato texto. A classe **BufferedReader** usa um objeto da classe **InputStreamReader**, o qual usa um objeto da classe **FileInputStream**. Essa última classe trabalha com bytes, a segunda classe converte os dados para *char* e a **BufferedReader** finalmente trabalha com grupos de caracteres (*string*).

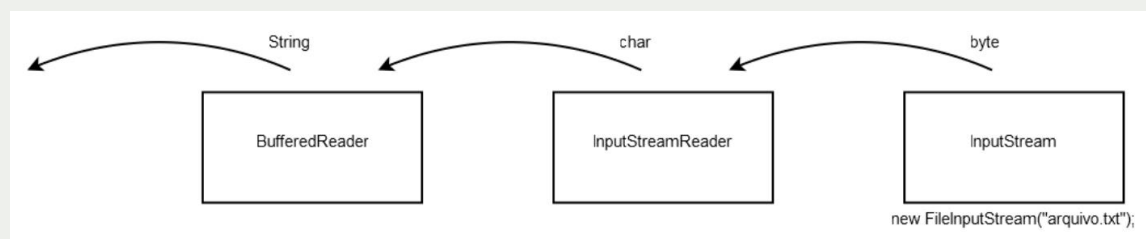


Figura 62 – Classes Java envolvidas no processo de leitura de arquivos do tipo texto.  
Fonte: Caelum (2022).

A Figura 63 mostra um exemplo de código das classes citadas acima, nele é acessado um arquivo chamado “arquivo.txt”, o qual se encontra no mesmo diretório do projeto. O algoritmo simplesmente imprime no console do computador todas as linhas desse arquivo.

```
1 FileInputStream arquivo = new FileInputStream("arquivo.txt");
2 InputStreamReader conversor = new InputStreamReader(arquivo);
3 BufferedReader dados = new BufferedReader(conversor);
4 String s;
5 do {
6     s = dados.readLine();
7     if (s == null)
8         break;
9
10    System.out.println(s);
11 } while (true);
12 dados.close();
```

Figura 63 – Exemplo de código das classes Java envolvidas no processo de leitura de arquivos do tipo texto. Fonte: próprio autor.

As linhas 1 até 3 (Figura 63) criam os objetos necessários para a leitura do arquivo. Repare que o objeto “dados” recebe como parâmetro o objeto “conversor”, esse último recebe como parâmetro o objeto “arquivo”.

Ainda no código da Figura 63, repare que ele utiliza o método “.readLine()” da classe *BufferedReader*. Esse método devolve um texto com a linha que foi lida do arquivo e muda o cursor de leitura para a próxima linha. Caso o final do arquivo seja alcançado, o método devolve o valor “null”. Esse valor nulo é útil para controlar o laço de repetição responsável pela leitura dos dados. Assim, a linha 6 do algoritmo é responsável por buscar a próxima linha de conteúdo do arquivo e armazena-lo na variável “s”. Se o valor da variável for justamente a palavra “null”, o laço de repetição é encerrado (linhas 7 e 8). Caso contrário, a linha 10 imprime o conteúdo e o laço de repetição continua sua execução, isso até que o fim do arquivo seja alcançado. A linha 12 libera o arquivo para uso de outras pessoas ou programas.

## 4.6 Gravando dados de arquivos texto

Para escrever em um arquivo do tipo texto, nós vamos fazer algo similar à leitura. Assim, a partir de uma sequência de palavras (*string*), o Java irá convertê-las em um grupo de caracteres (char) que por fim, serão convertidos para bytes. A Figura 64 mostra esse fluxo, iremos usar a classe **BufferedWriter** para escrever o texto no arquivo, mas essa classe irá usar a classe **OutputStreamWriter** para criar um fluxo de caracteres, essa última, irá utilizar a classe **OutputStream** para criar um fluxo de bytes.

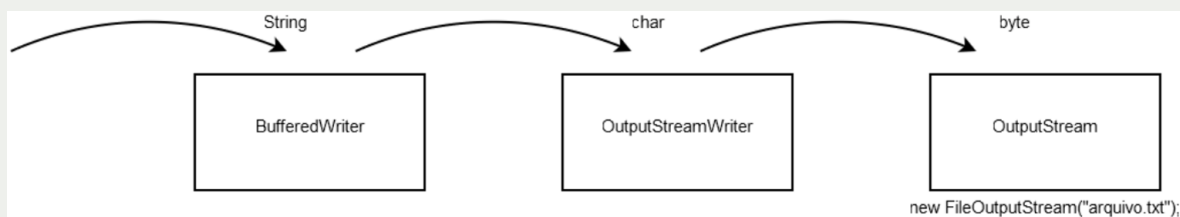


Figura 64 – Classes Java envolvidas no processo de escrita de arquivos do tipo texto.

Fonte: Caelum (2022).

A Figura 65 mostra um exemplo de código das classes citadas acima. O algoritmo simplesmente escreve no arquivo a palavra “Java”. As linhas 1 até 3 do algoritmo criam os objetos necessários para a gravação no arquivo. Repare que o objeto “dados” recebe como parâmetro o objeto “conversor”, esse último recebe como parâmetro o objeto “arquivo”. Repare na linha 1 que *FileOutputStream* recebeu um booleano como segundo parâmetro, ele indica se o programador quer reescrever o arquivo ou manter o que já estava escrito (append). O valor “true” significa justamente essa última opção.

Ainda no código da Figura 63, repare que ele utiliza o método “*.write()*” da classe *BufferedReader*. Esse método é responsável para, efetivamente, escrever algum texto no arquivo (linha 5). É importante reforçar que esse método não insere o caractere de quebra de linha. Para isso, você pode chamar o método “*.newLine()*”. Por fim, A linha 10 libera o arquivo para uso de outras pessoas ou programas.

```

1 FileOutputStream arquivo = new FileOutputStream("arquivo.txt", true);
2 OutputStreamWriter conversor = new OutputStreamWriter(arquivo);
3 BufferedWriter dados = new BufferedWriter(conversor);
4
5 dados.write("Java");
6 dados.close();
  
```

Figura 65 – Exemplo de código das classes Java envolvidas no processo de leitura de arquivos do tipo texto.

Fonte: próprio autor.



**Mídia digital:** Volte à sala virtual e assista ao vídeo “Manipulação de arquivos do tipo texto”, a ideia é reforçar os conceitos iniciais desse assunto tão importante.

## 4.7 Simplificando o acesso a arquivos do tipo texto

Se você achou complicado o processo de leitura e gravação de arquivos do tipo texto em Java. Nós temos uma boa notícia, as classes chamadas **FileReader** e **FileWriter**



do pacote **java.io** são atalhos para a leitura e escrita de arquivos texto. A Figura 66 mostra um exemplo de uso da classe *FileWriter*, a qual escreve dados em um arquivo texto. O método recebe como parâmetro um vetor de nomes e cada nome é escrito em uma linha do arquivo. Mais detalhes do código são explicados na própria figura.

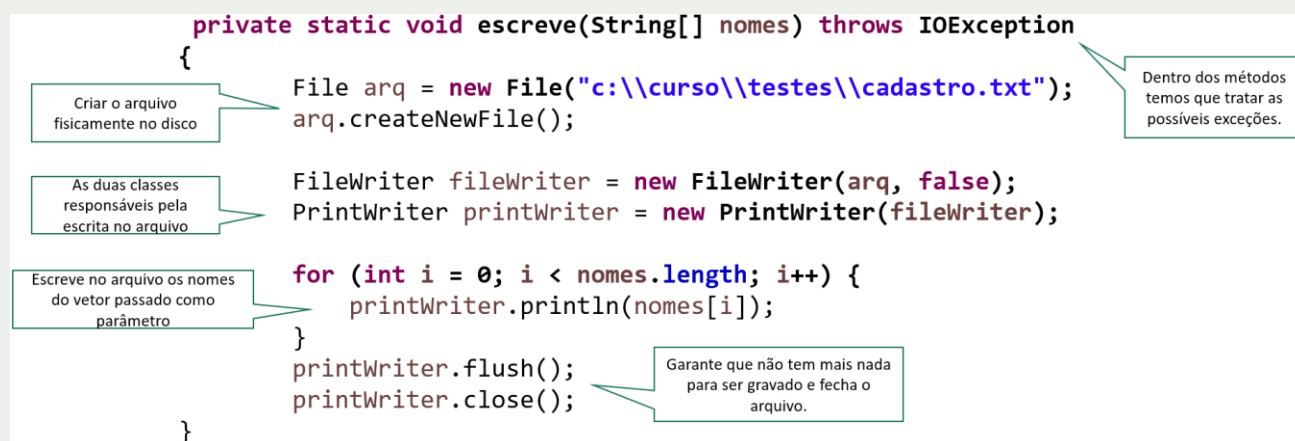


Figura 66 – Exemplo de código das classes *FileWriter* e *PrintWriter*.  
Fonte: próprio autor.

A Figura 67 mostra um exemplo de uso da classe *FileReader*, a qual lê dados em um arquivo texto. O método abre o arquivo `cadastro.txt` e imprime todas as suas linhas no console do computador. Mais detalhes do código são explicados na própria figura.

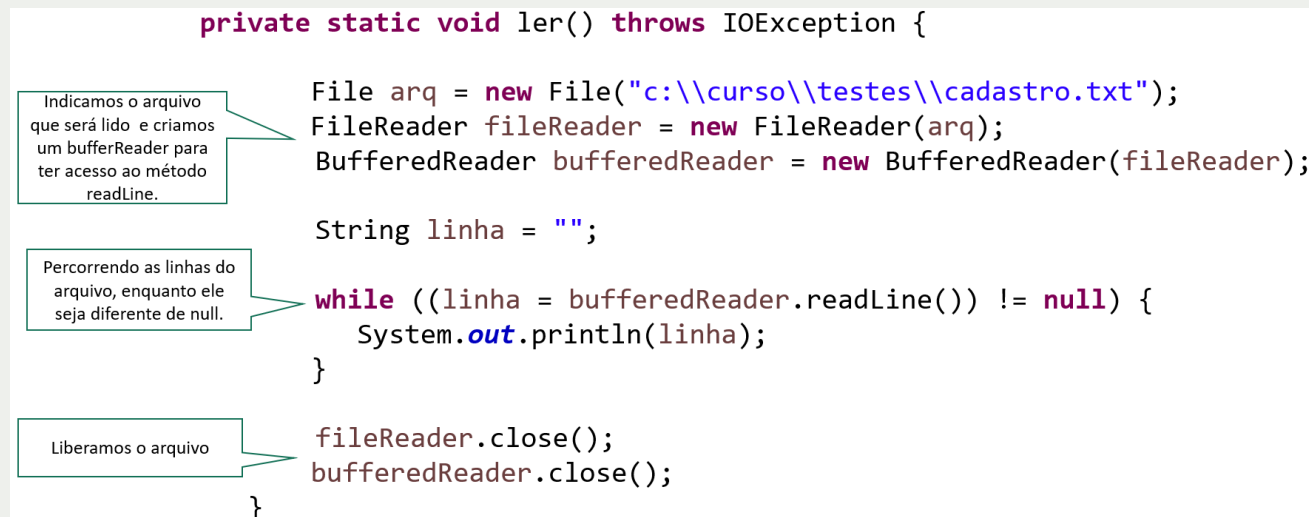


Figura 67 – Exemplo de código das classes *FileReader*.  
Fonte: próprio autor.

Como exemplo de uso, os métodos das Figuras 66 e 67 são chamados pelo trecho de código apresentado na Figura 68.

```

    public static void main(String[] args) throws IOException {
        String nomes[] = {"Johnny", "Jimmy", "Janes"};
        escreve(nomes);
        ler();
    }

```

Chamada dos métodos no main

Figura 68 – Chamada dos métodos de escrita e leitura de arquivos do tipo texto.

Fonte: próprio autor.



**Mídia digital:** Volte à sala virtual e assista ao vídeo “Simplificando a manipulação de arquivos do tipo texto”, a ideia é reforçar os conceitos iniciais desse assunto tão importante.



**Atenção:** No ambiente virtual, procure pela atividade chamadas “Questionário Final”. Essa atividade é avaliativa e a conclusão dela é necessária para a obtenção do seu certificado.

Concluimos assim, a nossa última semana de curso. Estudamos até aqui alguns dos conceitos mais avançados para a criação de algoritmos. Mais uma vez, assista aos vídeos propostos e leia o texto base do curso quantas vezes forem necessárias.

Espero que o curso tenha sido proveitoso e quero ver você no mercado de trabalho, o quanto antes. Mas lembre-se, esse foi só mais um passo para se tornar um profissional, seu estudo deve ser contínuo e assim, que dominar os conceitos apresentados aqui, procure novos desafios, a própria Plataforma +IFMG pode te ajudar a evoluir, consulte nossos outros cursos.

Parabéns pela conclusão do curso. Foi um prazer tê-lo conosco!

Boa Sorte!





## Referências

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Veneruchi de. Fundamentos da Programação de Computadores: Algoritmos, Pascal, C/C++ e Java. 2ª ed. São Paulo:, Pearson Education, 2008.

CAELUM, Ensino e Inovação. Java e Orientação a Objetos. Apostila digital disponibilizada pela empresa Caelum – Ensino e Inovação, 2020. Disponível em <https://www.caelum.com.br/apostilas>. Acesso em 11/06/2022.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: Teoria e Prática 1. Tradução da segunda edição [americana] Vandenberg D. de Souza. - Rio de Janeiro: Elsevier, 2002 - Reimpressão.

DEITEL, Paul J.; DEITEL, Harvey M. C: Como programar. 6ª ed. São Paulo: Prentice Hall, 2013.

FILHO, Clézio F. História da computação [recurso eletrônico]: O Caminho do Pensamento e da Tecnologia / Clézio Fonseca Filho. – Porto Alegre: EDIPUCRS 205 p., 2007.

MANZANO, J. A. N. G.; OLIVEIRA, J. G. de. Estudo Dirigido de Algoritmos. São Paulo. Editora Érica (Coleção PD). 1997.

SEBERTA Robert, Conceitos de Linguagens de Programação, ed. Bookman, 9a edição, 2011.



## Currículo do autor



**Bruno Ferreira** possui graduação em Ciência da Computação pelo Centro Universitário de Formiga (2003), pós graduação em Redes de Computadores pelo Centro Universitário do Sul de Minas (2004), mestrado em Modelagem Matemática e Computacional pelo CEFET-MG (2008) e doutorado em Ciência da Computação pela UFMG (2016). Tem experiência na área de Ciência da Computação, com ênfase em Métodos Formais, Linguagens de Programação e Engenharia de Software. Trabalhou no setor privado por 8 anos como programador e analista de sistemas. Atualmente é professor efetivo do Instituto Federal de Minas Gerais - (IFMG - Campus Formiga).

Currículo Lattes: <http://lattes.cnpq.br/9437251245138635>

Feito por (professor-autor)	Data	Revisão de <i>layout</i>	Data	Versão
Bruno Ferreira	14/06/2022	Viviane Lima Martins	19/06/2022	1.0



## Glossário de códigos QR (Quick Response)

		Mídia digital Apresentação do curso			Mídia digital Instalando os programas
		Mídia digital Introdução aos vetores			Mídia digital Percorrendo os vetores
		Mídia digital Exemplos de uso de vetores			Mídia digital Introdução às matrizes
		Mídia digital Percorrendo as matrizes			Mídia digital Tamanho da estrutura
		Mídia digital Exemplos com matrizes			Mídia digital Métodos – Introdução
		Mídia digital Métodos – Retorno de valores			Mídia digital Métodos – Tipos de passagem de parâmetros
		Mídia digital Introdução à manipulação de arquivos			Mídia digital Introdução à classe File
		Mídia digital Manipulação de arquivos binários			Mídia digital Manipulação de arquivos do tipo texto
		Mídia digital Simplificando a manipulação de arquivos do tipo texto			



## Plataforma +IFMG

### Formação Inicial e Continuada EaD



A Pró-Reitoria de Extensão (Proex), desde o ano de 2020, concentrou seus esforços na criação do Programa +IFMG. Esta iniciativa consiste em uma plataforma de cursos *online*, cujo objetivo, além de multiplicar o conhecimento institucional em Educação à Distância (EaD), é aumentar a abrangência social do IFMG, incentivando a qualificação profissional. Assim, o programa contribui para o IFMG cumprir seu papel na oferta de uma educação pública, de qualidade e cada vez mais acessível.

Para essa realização, a Proex constituiu uma equipe multidisciplinar, contando com especialistas em educação, *web design*, *design* instrucional, programação, revisão de texto, locução, produção e edição de vídeos e muito mais. Além disso, contamos com o apoio sinérgico de diversos setores institucionais e também com a imprescindível contribuição de muitos servidores (professores e técnico-administrativos) que trabalharam como autores dos materiais didáticos, compartilhando conhecimento em suas áreas de

atuação.

A fim de assegurar a mais alta qualidade na produção destes cursos, a Proex adquiriu estúdios de EaD, equipados com câmeras de vídeo, microfones, sistemas de iluminação e isolamento acústica, para todos os 18 *campi* do IFMG.

Somando à nossa plataforma de cursos *online*, o Programa +IFMG disponibilizará também, para toda a comunidade, uma Rádio Web Educativa, um aplicativo móvel para Android e IOS, um canal no Youtube com a finalidade de promover a divulgação cultural e científica e cursos preparatórios para nosso processo seletivo, bem como para o Enem, considerando os saberes contemplados por todos os nossos cursos.

Parafraseando Freire, acreditamos que a educação muda as pessoas e estas, por sua vez, transformam o mundo. Foi assim que o +IFMG foi criado.

O +IFMG significa um IFMG cada vez mais perto de você!

Professor Carlos Bernardes Rosa Jr.  
Pró-Reitor de Extensão do IFMG







Características deste livro:

Formato: A4

Tipologia: Arial e Capriola.

E-book:

1ª. Edição

Formato digital

