

LÓGICA DE PROGRAMAÇÃO

CURSO TÉCNICO EM INFORMÁTICA PARA INTERNET



Créditos

Equipe IFSC - Santa Catarina

Professores

Juliano Lucas Gonçalves
Andrino Fernandes
Glauco Cardozo
Marcos André Pisching

Equipe Designa Design Instrucional

Coordenação de Projeto

Cíntia Costa
Ester Konig

Coordenação de Design Instrucional

Emily Mercuri

Design Instrucional

Joyce Paola Mangrich

Design Gráfico

Ayrin Barboza

Revisão ortográfica

Priscila Verçosa

**Este trabalho está licenciado
sob CC BY-NC 4.0** 



Apresentação

Caro estudante, estamos iniciando uma Unidade Curricular muito importante para a sua formação profissional. É a partir da Lógica de Programação que você terá condições de produzir softwares de qualquer natureza.

Atualmente os softwares estão presentes em muitas atividades no nosso dia a dia. Se nos atentarmos, veremos o quanto eles nos influenciam direta e indiretamente. O simples fato de ter em mãos um smartphone já nos proporciona o acesso a uma série de informações, podemos nos comunicar, nos orientar geograficamente, gerenciar contas bancárias, realizar compras, utilizar jogos eletrônicos e muito mais.

E com as pessoas jurídicas, como indústrias, empresas, lojas, escolas, profissionais liberais, entre outros, ocorre da mesma forma! Cada uma terá diferentes necessidades relacionadas aos softwares. Seja para: controlar seus funcionários, clientes, fornecedores; estabelecer uma mídia social ou marketing; controlar a contabilidade; gerar relatórios entre muitos outros processos. Além do mais, é redundante ‘dizer’ que essa tecnologia tem crescido e evoluído continuamente, o que estabelece uma necessidade imensa de profissionais relacionados com a tecnologia da informação, em especial, de desenvolvedores de software e, mais especificamente, de desenvolvedores web.

Vale destacar que os profissionais que desenvolvem sistemas, aplicativos e softwares ocupam, atualmente, pelo menos, 40% das vagas em TI. E há outras áreas relacionadas que aumentam ainda mais as oportunidades, como administradores de banco de dados, engenheiros de softwares e web designers. Sendo assim, seja bem-vindo(a)! Será um prazer acompanhá-lo(a) em mais uma etapa neste fascinante mundo dos softwares, da conectividade e da informação.



No Estado, a projeção é a de ultrapassar a marca de 100 mil vagas até o final de 2025, incremento de 44,3% ante às atuais 76,6 mil ativas. Os números compõem o Mapeamento de Vagas 2023, realizado pela Associação Catarinense de Tecnologia (Acate) e execução do Instituto Mapa.
(Pessotto, 2023, on-line).



Atente-se para a realização de todas as atividades, planeje os seus estudos, determine horários e locais adequados, organize estudos com seus colegas e não deixe acumular conteúdo, viu?

Sucesso e boas aulas!

Objetivos

- Compreender, interpretar e desenvolver algoritmos e programas com linguagem de programação.
- Conhecer e aplicar técnicas para o desenvolvimento do raciocínio lógico na programação.
- Analisar, entender e otimizar código de programação.

Sumário



1. Introdução à Lógica de Programação	6
2. Algoritmo	9
3. Expressões e Operadores	28
4. Introdução a Linguagem Java	36
5. Estruturas de Controle de Seleção	58
6. Estruturas de Controle de Repetição	89
7. Variáveis compostas homogêneas	111
8. Subprogramas	133
Finalizando	142
Referências	143

1. Introdução à Lógica de Programação

O que é Lógica? O que significa estudar Lógica? E Linguagem de Programação? Como um computador “entende” e “roda” os programas? Não se preocupe! Esses e outros conceitos relacionados serão respondidos à medida que estudarmos e abordarmos os novos conteúdos.

Para começar a nossa reflexão, André Forbellone e Henri Eberspächer (2005, p. 1) explicam que:

66

A lógica, em geral, trata da correção do pensamento. Como filosofia, ela procura saber por que pensamos assim e não de outro jeito. Como arte ou técnica, ela nos ensina a usar corretamente as leis do pensamento. Pode-se dizer também que a lógica é a arte de pensar corretamente. Visto que a forma mais complexa do pensamento é o raciocínio, a lógica estuda ou tem em vista a “correção do raciocínio”. Pode-se ainda dizer que a lógica tem em vista a “ordem da razão”. Por isso, a lógica ensina a colocar “ordem no pensamento”.

99

Mas, antes de começarmos a programar de fato, precisaremos entender o conceito de algoritmo e como construí-lo. Afinal, são conceitos que fundamentam o conhecimento sobre a programação de computadores.

O que é lógica?

A palavra **lógica** está normalmente relacionada com o modo de pensar de um indivíduo em termos de racionalidade e coerência. Apesar de ela ser contextualizada e estar frequentemente associada à matemática, a lógica também tem a sua

aplicação em outras ciências e, principalmente, nas ações individuais de cada pessoa.

No campo da filosofia, a lógica visa a estudar a estrutura formal dos enunciados (proposições) e as suas regras. Portanto, a lógica serve para se pensar corretamente, ou seja, é uma ferramenta do “correto pensar”.



“Lógica” tem origem na palavra grega *logos*, que significa “razão, argumentação ou fala”. A ideia de falar e argumentar pressupõe que o que está sendo dito tem um sentido para aquele que ouve. Esse sentido fundamenta-se na estrutura lógica. Quando algo “tem lógica”, quer dizer que faz sentido, é uma argumentação racional.

Confira um exemplo de lógica proposicional:

Todo cachorro é um mamífero.

Todo mamífero é um animal.

Portanto, todo cachorro é um animal.

O Japão é um país do continente asiático.

Todos os japoneses são do Japão.

Logo, todos os japoneses são asiáticos.

Os exemplos apresentados representam um argumento composto por duas premissas e uma conclusão.

Agora, acompanhe mais detalhes sobre a definição da lógica de programação.

O que é lógica de programação?

A lógica de programação é a forma como as ações serão conduzidas ou realizadas por um programa de computador. Toda programação apresenta um encadeamento lógico para que as instruções (ou ações) descritas possam executar os comandos determinados para a solução de um problema. Nesse sentido, quem programa é responsável por compreender essa lógica e traduzi-la de forma eficiente para a máquina.



Você deve estar muito curioso(a) para saber como os computadores fazem para executar essa “sequência de instruções” e como conseguiremos habilidades para que eles (os computadores) nos entendam, não é mesmo? Para isso, é muito importante, primeiramente, entender a relação que existe entre a lógica humana e a lógica de programação, pois, em muitos momentos, perceberemos uma estreita relação entre elas e, assim, teremos o entendimento de por que e como as linguagens de programação foram criadas e continuam evoluindo.

Então, para ser um bom programador, será preciso, principalmente, conhecer linguagens de programação, certo? Ops! Não é bem assim... Para ser um bom, ou melhor, um excelente programador, você precisa ter competência e habilidade em lógica de programação. Conhecer as linguagens é igualmente importante para a experiência e formação profissional, mas a lógica de programação é o “campo” a ser preparado para que a “colheita” seja farta e contínua.

É até possível aprender a lógica da programação enquanto se aprende uma linguagem de programação, mas a evolução do aprendizado tende a ser bem mais lenta.

Com a lógica de programação, você vai adquirir conhecimentos e técnicas necessários para aprender qualquer linguagem de programação. Lembre-se disso e vamos em frente!

2. Algoritmo

Quando se fala em algoritmo, muitas pessoas pensam rapidamente em computadores, tecnologia e até mesmo códigos difíceis de serem compreendidos. No entanto, o conceito e a aplicação são bem mais simples do que parecem. Embora tenha designação desconhecida, é importante ressaltar que usamos constantemente os algoritmos em nosso cotidiano.



Fonte: JPortugol (2011).

Basicamente, o algoritmo é uma sequência de passos ordenados logicamente para atingirmos um objetivo.



Um algoritmo é uma sequência finita de ações executáveis que visam obter uma solução para um determinado tipo de problema. Segundo Dasgupta, Papadimitriou e Vazirani, "Algoritmos são procedimentos precisos, não ambíguos, padronizados, eficientes e corretos. (Wikipédia, 2024, on-line)."



Para termos uma ideia melhor, vamos indicar uma ação muito comum: tomar um banho no chuveiro. Pense em cada passo que deve ser seguido para que o banho aconteça... Agora vamos formalizar este processo, ou seja, vamos descrever cada ação que deverá ser realizada:

- 1º passo: tirar a roupa.
- 2º passo: abrir o chuveiro.
- 3º passo: molhar o corpo.
- 4º passo: ensaboar-se.
- 5º passo: enxaguar o corpo.
- 6º passo: fechar o chuveiro.
- 7º passo: secar-se.

É importante perceber que:

01

Temos uma sequência ordenadas de passos (ou ações).

02

Os passos estão ordenados logicamente.

03

A conclusão do algoritmo implicará a obtenção do objetivo definido.

Se seguirmos um passo após o outro, atingiremos nosso objetivo? Sim! A ordem lógica é fundamental para o sucesso do algoritmo. Não é difícil perceber que, possivelmente, se a ordem for alterada, alguma coisa errada ou estranha poderá comprometer o objetivo, que é ter o corpo limpo após o banho. Da mesma forma, pode ser que mudar a ordem não afete o resultado. Isso significa que o mesmo objetivo pode ter caminhos (algoritmos) diferentes.

Bem! Então você já deve ter percebido que tudo (ou quase tudo) que fizemos é uma ação racional e lógica que nos leva à obtenção de um resultado, certo? Podemos citar vários outros exemplos: uma receita de bolo, um manual que orienta sobre a montagem um móvel, a forma como calcular o valor do consumo de energia elétrica etc. Mas, como o nosso objetivo é a programação de computadores, nossos exercícios e problemas serão direcionados para temas que lembrem ou sejam pertinentes à programação.

Então, o algoritmo serve como modelo para programas, pois usa uma linguagem intermediária à linguagem humana e às linguagens de programação, sendo uma boa ferramenta na validação da lógica de tarefas a serem automatizadas.

Os algoritmos servem para representar a solução de qualquer problema, mas no caso do processamento de dados, eles devem seguir as regras básicas de programação para que sejam compatíveis com as linguagens de programação.

Confira, a seguir, alguns conceitos que são citados em literaturas especializadas sobre o algoritmo. Segundo Knuth (1977, p. 4), algoritmo é:

 Um conjunto finito de regras que provê uma sequência de operações para resolver um tipo de problema específico.



Já Tremblay e Soreson (1984, p. 7) descrevem-no da seguinte maneira:

 Sequência ordenada, e não ambígua, de passos que levam à solução de um dado problema.



Por fim, Ferreira (1999, p. 33) informa que este é um:

 Processo de cálculo, ou de resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições, as regras formais para a obtenção do resultado ou da solução do problema.



Vejamos as características de um algoritmo:

-  Ter um início.
-  Ter um fim.
-  Não dar margem à dupla interpretação (não ter duplo sentido).
-  Ter a capacidade de receber dado(s) de entrada do mundo exterior.
-  Poder gerar informações de saída para o mundo externo ao do ambiente do algoritmo.
-  Ser efetivo, ou seja, todas as etapas especificadas no algoritmo devem ser alcançáveis em um tempo finito.

Perceba, pelas características, que, a partir de agora, os nossos algoritmos terão uma aparência, e seus objetivos lembrarão programas. E, mais tarde, você irá transformá-los em programas! Será muito legal!

Atente-se para o próximo tópico sobre as técnicas para construção de algoritmos e entenda as diferenças. E se você acha que ficaremos no campo teórico, prepare-se, porque vamos utilizar algumas ferramentas para testar e entender os algoritmos.

Formas de representação de um algoritmo

Vamos iniciar ressaltando que a importância do algoritmo está no fato de termos que especificar uma sequência de passos lógicos para que o computador possa executar uma tarefa qualquer, pois ele por si só não tem vontade própria, faz apenas o que mandamos. Com uma ferramenta algorítmica, podemos conceber uma solução para um dado problema, independentemente de uma linguagem específica e até mesmo do próprio computador.



Neste sentido, destacam-se as técnicas para construção de algoritmos, que são:

- Descrição narrativa.
- Fluxograma.
- Pseudocódigo.

Conheça cada uma delas a seguir.

Descrição narrativa

Essa forma de algoritmo é a mais simples e informal, é a descrição do passo a passo das tarefas que devem ser executadas. O algoritmo “Tomar um banho no chuveiro”, do tópico anterior, já serviu como um exemplo simplificado de uma ação do nosso cotidiano, assim como os exemplos citados.

Vamos agora mostrar um novo problema, mas já pensando nele como um futuro programa. Vamos começar com um problema simples: qual é o algoritmo para

realizar a média em dois valores numéricos?

É normal, neste início, haver alguma dificuldade para formalizar algumas tarefas. Uma regra valiosa é identificar de quais dados o algoritmo precisa para dar prosseguimento às próximas ações, e o que o algoritmo precisa fornecer como resultado.

Desta forma, segundo o enunciado, o algoritmo precisará de dois dados que são os valores numéricos, sem eles, nada será possível. Acompanhe estes passos:

- **1º Passo:** obter o primeiro número.
- **2º Passo:** obter o segundo número.
- **3º Passo:** realizar a soma dos dois números obtidos.
- **4º Passo:** realizar a divisão por 2 da soma anterior.
- **5º Passo:** apresentar o resultado da divisão.

Você pode realizar a execução desse algoritmo com a ajuda de outra pessoa. Peça a ela para informar dois números (passos 1 e 2) e, na sequência, você realiza os passos 3 e 4, então, informe o resultado a ela. Será muito próximo do que um programa irá realizar. No caso, você fez o papel do computador: solicitou as entradas, realizou os cálculos e apresentou o resultado.

Fluxograma

O fluxograma é a representação por meio de símbolos gráficos que mostram a sequência de execução, é uma das maneiras possíveis de representar os algoritmos.

Existem símbolos padronizados para início, entrada de dados, cálculos, saída de dados, fim e outras funções.

Portanto, um fluxograma é um diagrama que descreve um processo, sistema ou algoritmo de computador. É amplamente utilizado em várias áreas para documentar, estudar, planejar, melhorar e comunicar processos complexos por meio de diagramas claros e fáceis de entender.

66

O termo fluxograma designa uma representação gráfica de um determinado processo ou fluxo de trabalho, efetuado geralmente com recurso a figuras geométricas normalizadas e as setas unindo essas figuras geométricas. Através desta representação gráfica é possível compreender de forma rápida e fácil a transição de informações ou documentos entre os elementos que participam no processo em causa. (Wikipédia, 2022, on-line).

99

A construção de fluxogramas pode ser desenvolvida em uma folha de papel, em algum editor disponível ou até mesmo em ambiente em que podemos entender claramente as suas ações, o armazenamento de dados e a execução – que será o nosso caso. Mas, antes disso, é importante saber que cada ação será representada por uma figura, e a sequência de ações estará conectada por uma seta que indicará a sua ordem.

Nas suas leituras sobre fluxogramas, será comum ver representações diferentes para uma mesma ação, podendo ser parecidas, mas, de um modo geral, de fácil percepção.

Para tornar o processo de aprendizagem mais lúdico, adotaremos o ambiente **fluxolab.app** - que foi desenvolvido pelo professor Roberto Wanderley da Nóbrega

do IFSC – Campus São José. Assim, poderemos desenvolver alguns fluxogramas e executá-los entendendo com clareza todo o processo, pois cada ação será visualizada, assim como os dados envolvidos nesta relação.

Os símbolos (ações) mais comuns e as formas que os utilizaremos são:

Início e fim

Determinam o início e o término do fluxograma. Haverá apenas um Início e um Fim.

Entrada

Ação que determina a obtenção de um dado.

Cálculo

Processo que realiza a operação (normalmente matemática) e a atribuição do resultado em algum repositório.

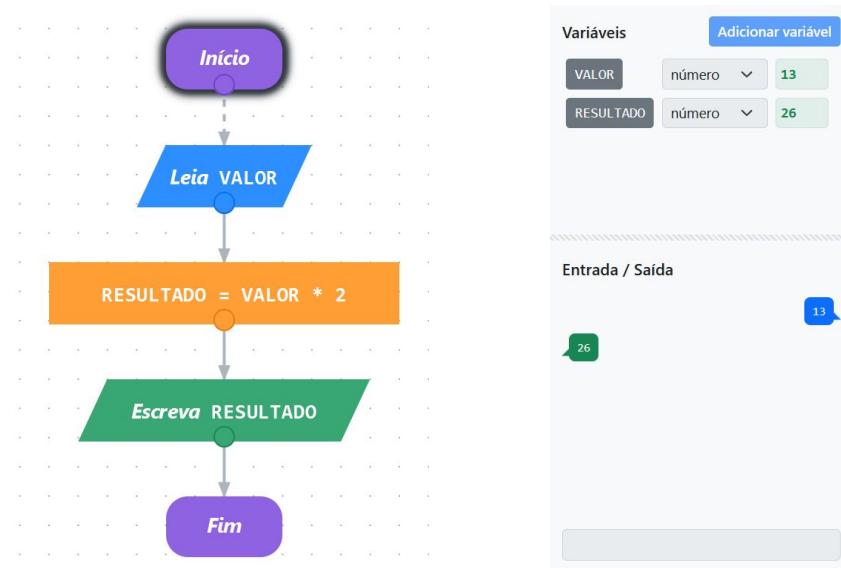
Saída

Ação que apresenta um resultado ou uma mensagem.

Decisão

É um ponto no fluxograma que decide por um de dois caminhos possíveis.

Veja o primeiro exemplo. O objetivo é obter um número e mostrar o seu dobro:



Fonte: Elaborada pelos autores (2024).

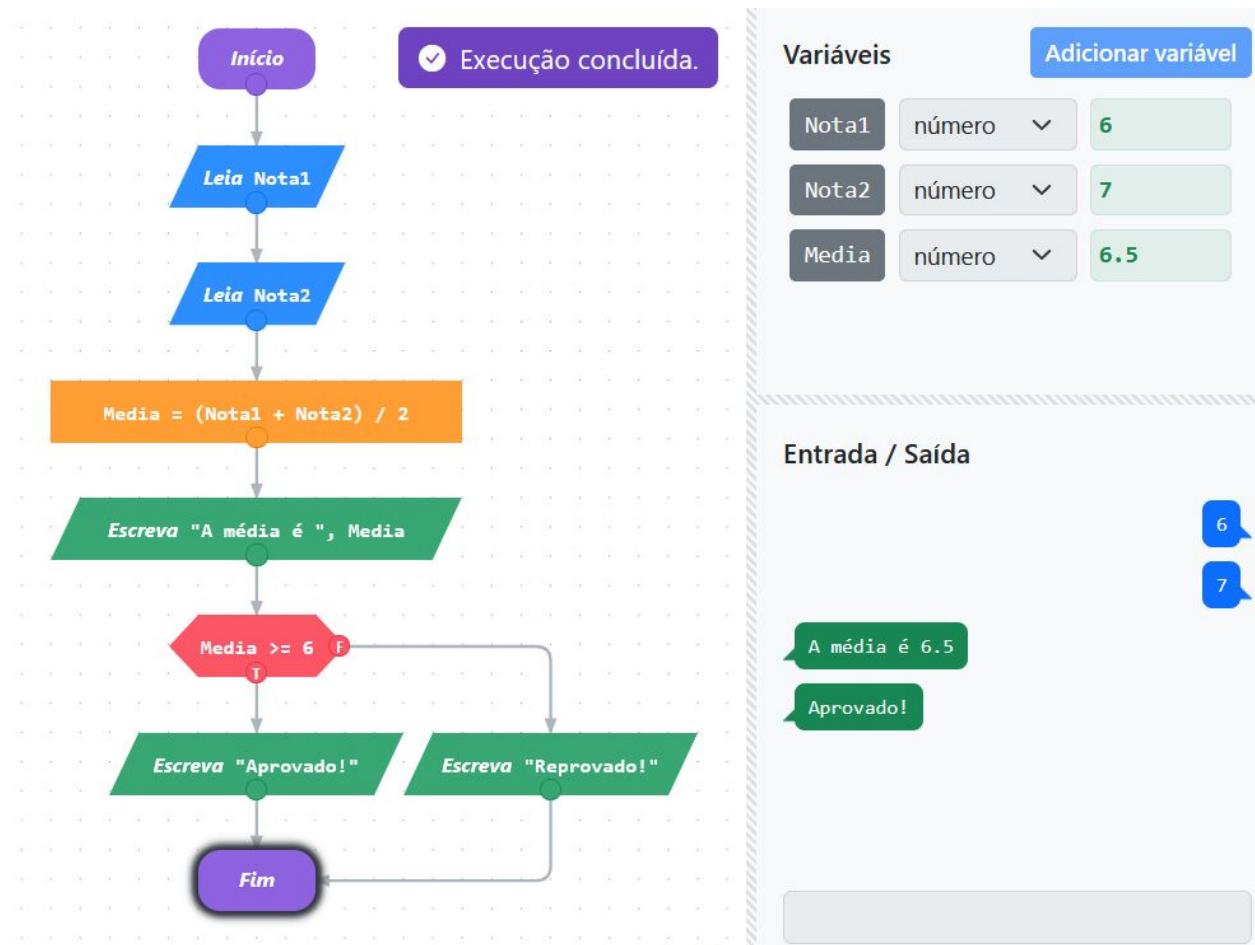
No ambiente **fluxolab.app**, você terá uma seção chamada de “Variáveis”, local em que os dados serão armazenados, no caso, VALOR e RESULTADO, que devem ser criados; e, mais abaixo, a seção “Entrada / Saída” é o local em que acontecerão os resultados das ações do nosso fluxograma.



O termo “Variável” será um conceito muito comum na programação. Ela funciona como um repositório para um dado onde a referência para sua utilização é o seu nome. As ações mais comuns que envolvem variáveis são: a de levar um dado para ser armazenado, que pode ser por meio de uma ação de entrada ou de atribuição, ou a de recuperar um dado, quando acontece uma instrução de saída ou quando ela se encontra em uma expressão.

Observe que, na ação de entrada, ou seja, no VALOR, foi digitado 13, que foi levado ao repositório/variável VALOR. Na sequência, veio o processo do Cálculo, que realizou a operação matemática $VALOR * 2$ e atribuiu o resultado 26 no repositório/variável RESULTADO. E, por último, a ação de saída que apresentou o resultado 26.

No segundo exemplo, utilizaremos uma Decisão para ilustrar como acontece a possibilidade de desvio de fluxo. Veja a seguir: calcular a média de duas notas, mostrá-la e informar se a média é de aprovação ou reprovação. Vamos considerar que a média para aprovação é 6.



Fonte: Elaborada pelos autores (2024).

O fluxograma mostra:

- A entrada das duas notas: Nota1 e Nota2.
- O cálculo da média e sua atribuição em Media.
- A apresentação da média.
- A Decisão, em que temos uma expressão lógica (Media \geq 6). Quando o fluxo chega na Decisão, o conteúdo de Media será comparado com 6. Dependendo do conteúdo, o fluxo tomará o caminho do T, que indica verdadeiro (True), ou irá tomar o caminho do falso (False).
- Se a Decisão for verdadeira, a mensagem “Aprovado!” será exibida, senão, a mensagem “Reprovado!” será exibida.

Perceba que no teste realizado foram digitadas as notas 6 e 7, respectivamente, o que gerou à média 6.5 e a mensagem “Aprovado!”.

Neste exemplo, tivemos três novidades:

- 1 - A expressão lógica na Decisão - que veremos de maneira mais completa no Tópico 3: “Expressões e Operadores”.
- 2 - A própria Decisão, neste caso, utilizada para selecionar o caminho a seguir, que será vista profundamente nos Tópicos 5 e 6: “Estrutura de Controle de Seleção” e “Estrutura de Controle de Repetição”.
- 3 - Um novo elemento na saída, o Literal.



O Literal é utilizado para exibir um texto ou uma mensagem que deverá estar entre aspas duplas (“”). Veja que no comando <Escreva “A média é ”, Media>, temos o Literal (“A média é ”) e a Variável (Media). O Literal sempre apresentará que está dentro das aspas e a Variável sempre mostrará o seu conteúdo. No caso demonstrado, a mensagem resultante exibida foi “A média é 6.5”.



Deu bug?

Agora é com você! Considerando o fluxograma anterior, quais serão as mensagens exibidas quando a Nota1 receber 4.5 e a Nota2 receber 5?

Agora, conheça a próxima técnica: pseudocódigo. Vamos lá?

Pseudocódigo

Entre as técnicas de construção de algoritmos, o pseudocódigo é a que mais se aproxima de um programa. Como não é um código – o que seria um programa na área da computação – e o fato de termos o sufixo pseudo (o que parece, mas não é), caracteriza o pseudocódigo com um conjunto de instruções muito parecidas com um programa.

As instruções utilizadas no pseudocódigo, diferentemente das linguagens de programação em que se utiliza o inglês, são definidas em português justamente para não haver nenhuma dificuldade no seu entendimento. Em função disso, essa pseudolínguagem também é conhecida como Portugol.



Pseudocódigo é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples (nativa a quem o escreve, de forma a ser entendida por qualquer pessoa) sem necessidade de conhecer qualquer sintaxe de qualquer linguagem de programação livre de contexto... Os livros sobre a ciência de computação utilizam frequentemente o pseudocódigo para ilustrar seus exemplos, de forma que todos os programadores possam entendê-los, independentemente da linguagem que utilizem. (Wikipédia, 2023, on-line).



Um fator interessante na utilização do fluxolab.app para o desenvolvimento de fluxogramas, que vimos anteriormente, é que as instruções ficam descritas dentro das figuras, o que ajudará na compreensão e construção do pseudocódigo.

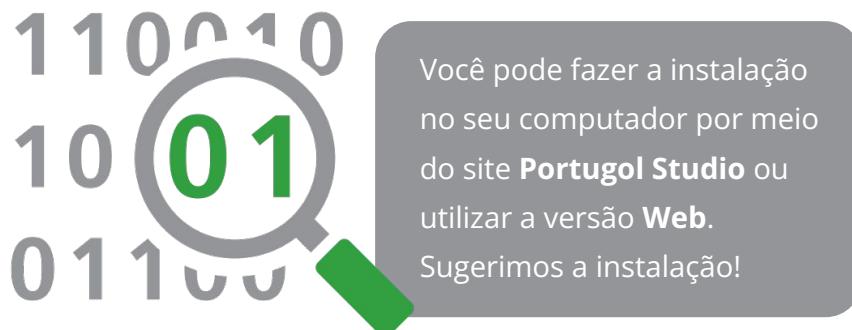
Assim como no fluxograma, vamos apresentar um novo ambiente em que poderemos desenvolver nossos pseudocódigos. Trata-se do Portugol Studio da Universidade do Vale do Itajaí (Univali), um ambiente de programação que foi desenvolvido especialmente para os iniciantes da programação. Então, nessa pseudolínguagem, será possível desenvolver nossos 'programas', testá-los e entendê-los.

Mas é importante ressaltar que quando formos testar a sua execução, o programa deverá estar escrito corretamente. Caso não esteja, o **compilador** do ambiente apresentará mensagem(ns) de erro(s).

Cyberpedia

Compilador: um processo (ou mesmo um programa) que analisa todo o programa desenvolvido e que o executa quando toda a codificação (programa) estiver sem erros na sua escrita.

Antes de partirmos para a construção do nosso programa em Portugol Studio, é importante saber que, para utilizá-lo:



Vejamos o primeiro exemplo, que é o mesmo utilizado no primeiro fluxograma: cálculo de dobro de um número.

```
1/* Programa para
2    calcular e mostrar
3    o dobro de um número lido. */
4programa
5{
6    funcao inicio()
7    {
8        inteiro Valor, Resultado // declaração das variáveis
9        leia (Valor)           // comando de entrada
10       Resultado = Valor * 2 // comando de atribuição
11       escreva (Resultado)  // comando de saída
12    }
13}
```

Fonte: Elaborada pelos autores (2024).

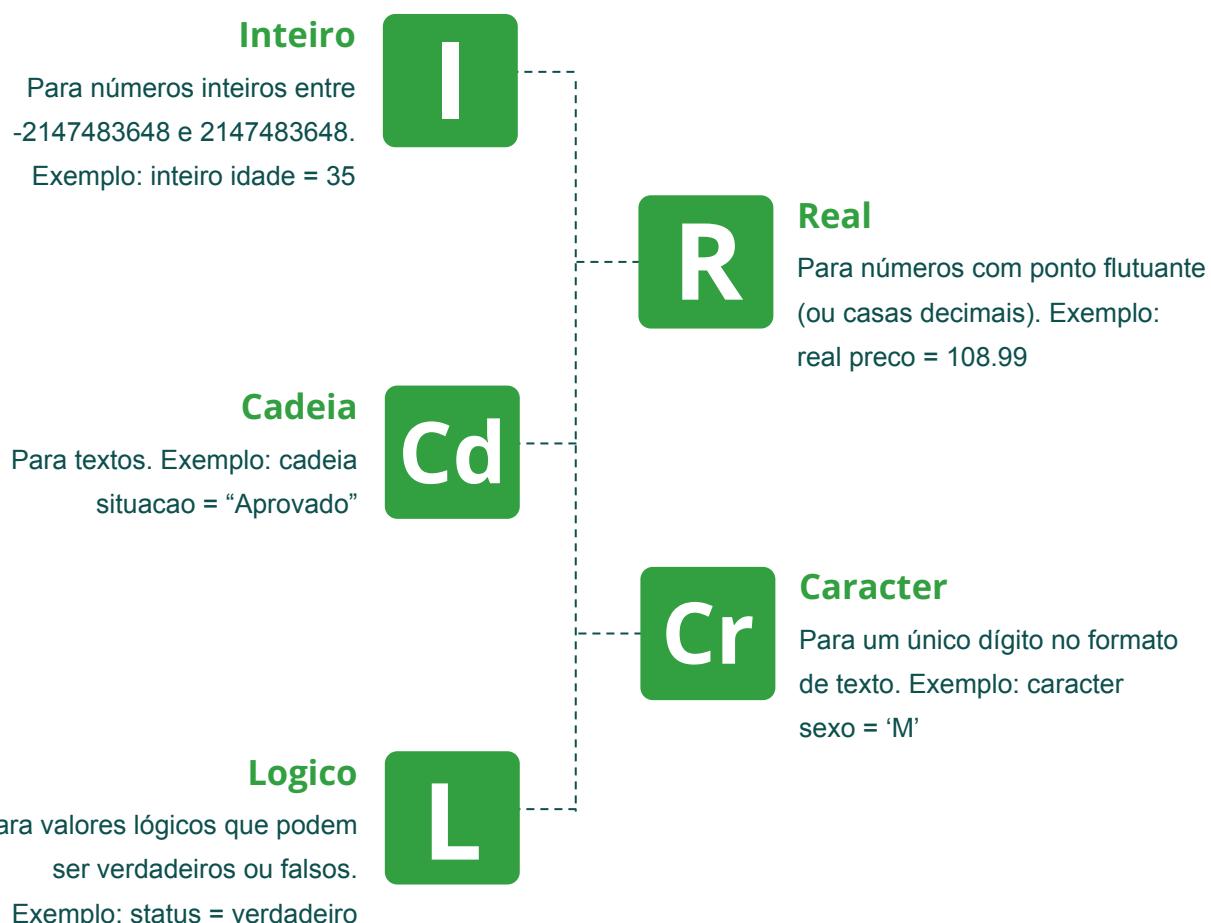
Atente-se para que a produção dos nossos programas aconteça dentro do bloco `inicio()` (trecho entre as chaves). Essa estrutura básica precisa ser preservada.

Uma novidade na codificação de programas é a utilização de comentários, que são descrições que podem ajudar no entendimento do programa ou de alguma outra informação. Os comentários são os trechos com `//`, que é um comentário de linha, ou o bloco `/* */`, que é o comentário para múltiplas linhas. Os comentários são ignorados quando são executados. Na linguagem Java (entre outras), que

utilizaremos, os comentários poderão ser utilizados da mesma forma.

Sobre o exemplo, veja que no pseudocódigo temos uma instrução no início que define o tipo das variáveis que serão utilizadas, também conhecida como declaração de variáveis. Então, declarar uma variável significa definir o seu tipo e o nome que ela terá. Internamente, na memória principal do computador, o programa reservará um espaço na memória principal para armazenar um dado, respeitando o tipo estabelecido. Você deve ter lembrado que já vimos as variáveis no Fluxograma. Lá, também, pelo ambiente do **Fluxolab**, um tipo pode ser definido. Essa relação é muito positiva, o que ajuda o nosso entendimento.

Vamos, então, conhecer os tipos de variáveis no Portugol Studio:



Mas, para utilizarmos variáveis, não basta reconhecer os seus tipos. Você já percebeu que elas têm uma identificação, que é a forma para referenciá-las. Então,

será importante sabermos como definir essa 'identificação'. Vejamos as regras:

1 - Não pode utilizar símbolos especiais (com exceção do underline (_)) como: *, @, !, ç, ã, -, o espaço em branco, +, :, ... Basicamente, são permitidas as letras maiúsculas e minúsculas, os números e o underline.

Exemplo:

Preço do Produto (errado!);

Preco_do_Produto (correto!).

2 - Não pode começar com número, ou seja, só pode começar com uma letra ou o underline.

Exemplo:

1a_nota (errado!);

nota1 ou nota_1 ou _n1 (correto!).

3 - Não pode ser palavra reservada. As palavras reservadas são as "palavras" que fazem parte do vocabulário da linguagem, como os comandos ou parte dele.

Exemplo:

inteiro (errado! - é uma palavra que identifica uma variável como tipo inteiro);

falso (errado! - é uma palavra utilizada como valor de uma variável lógica);

leia (errado! - é uma palavra utilizada no comando de entrada).

Nota 1

1

Essas mesmas regras serão utilizadas posteriormente quando trabalharmos com a linguagem de programação Java.

Nota 2

2

O Portugol Studio, assim como o Java, é *case sensitive*, ou seja, diferencia letras maiúsculas e minúsculas. Isso significa que não podemos declarar uma variável identificada como Valor e referenciá-la como valor – veja que a diferença está na letra V.

Vamos agora estabelecer a relação das principais ações já conhecidas no fluxograma e indicar a sintaxe, ou seja, a maneira correta de escrevê-las. São elas:

- a) Entrada: leia (<variável>)

Exemplo: leia (numero)

- b) Atribuição: <variável> = <constante>

<variável> = <variável>

<variável> = <expressão>

Exemplos: A = 5

atribuição de uma constante

B = A

atribuição de uma variável

Media = (Nota1 + Nota2) / 2

atribuição de uma expressão matemática

sinal = A >= B

atribuição de uma expressão lógica

Frase = A + " é igual a " + B

atribuição de uma expressão literal

Nota: veremos detalhadamente as expressões no próximo tópico.

- c) Saída: escreva (<variável e/ou literal e/ou expressão>)

Exemplo: escreva ("A média do aluno ", nome, " é ", (n1+n2)/2, "\n").

Nota: O \n indica uma quebra de linha que deverá estar dentro de um literal, por isso: "\n".

- d) Decisão: é utilizada para decidir por um de dois caminhos possíveis. Ela será vista detalhadamente nos Tópicos 5 e 6, quando estudaremos as estruturas de controle de seleção e de repetição.

Mas, para termos uma ideia básica, considerando que vimos uma situação no fluxograma, vamos mostrar como é a decisão utilizando a estrutura de controle de seleção SE no Portugol Studio.

SE simples

Sintaxe:

```
se (<condição>)
{
    <comando(s)> // realiza este bloco
        // quando a <condição>
        // for verdadeira
}
```

Obs.: quando a <condição> for falsa, nada acontecerá

Exemplo:

```
se (Media >= 6)
{
    escreva("Aluno(a) aprovado!")
}
```

SE composto

```
se (<condição>)
{
    <comando(s)> // realiza este bloco
        // quando a <condição>
        // for verdadeira
}
senao
{
    <comando(s)> // realiza este bloco
        // quando a <condição>
        // for falsa
}
```

Exemplo:

```
se (Media >= 6)
{
    escreva("Aluno(a) aprovado!")
}
senao
{
    escreva("Aluno(a) reprovado!")
}
```

Fonte: Elaborada pelos autores (2024).

Para uma ilustração prática, vamos aproveitar o mesmo exemplo utilizado no fluxograma, que trata do cálculo da média aritmética de um aluno. Incluiremos a entrada do nome do aluno.

```

1  programa
2  {
3      funcao inicio()
4      {
5          // declaração de variáveis
6          cadeia Nome
7          real Nota1, Nota2, Media
8          // entrada de dados
9          escreva("Digite o nome do aluno: ")
10         leia(Nome)
11         escreva("Digite a primeira nota: ")
12         leia(Nota1)
13         escreva("Digite a segunda nota: ")
14         leia(Nota2)
15         // cálculo da média
16         Media = (Nota1 + Nota2) / 2
17         // apresenta a média
18         escreva("A média de ", Nome, " é ", Media, "\n")
19         // testa a média e escreve uma das situações
20         se (Media >= 6)
21         {
22             escreva("Situação: Aprovado!")
23         }
24     senao
25     {
26         escreva("Situação: Reprovado!")
27     }
28 }
29 }
```

Fonte: Elaborada pelos autores (2024).

A utilização do comando “escreva”, na entrada de dados, indica e orienta sobre o dado em questão.

Na sequência, veja uma apresentação dentro do próprio ambiente da saída, que foi gerada pelo programa:

```

Digite o nome do aluno: João
Digite a primeira nota: 8
Digite a segunda nota: 7.7
A média de João é 7.85
Situação: Aprovado!
```

Fonte: Elaborada pelos autores (2024).

Agora, atente-se às listas de exercícios e atividades disponibilizadas no ambiente virtual. Elas serão muito importantes para a evolução do seu conhecimento lógico-computacional.

Para finalizar, acompanhe algumas boas práticas no desenvolvimento de programas que serão importantes na sua vida de programador:

01 Nomes

Atribua nomes às variáveis que tenham relação com o dado e que não sejam extensas. Cuidado com as siglas!

02 Organização e clareza na redação do programa

Utilize da indentação, que são os recuos dos comandos que estejam dentro de um bloco/uma estrutura para facilitar a leitura visual. Além disso, é importante a boa distribuição no fluxo dos comandos.

03 Comentários

Utilize comentários que realmente ajudem, especialmente quando os programas são extensos e/ou complexos e, principalmente, se forem compartilhados com outros programadores (equipe de desenvolvimento).

04 Otimização de código

É um processo por meio do qual fazemos um programa ser mais eficiente, com um número menor de passos e com um número reduzido de memória. Não é algo com que você precisa se preocupar neste momento, mas isso será discutido durante o curso, no desenvolvimento e nas reflexões de nossas soluções.

05 Funcionalidade

Atenção à funcionalidade do programa, pois erros de ordem lógica (não identificados pelo compilador) podem acontecer. Verifique se os resultados estão corretos, faça testes com dados diferentes e nunca deixe nenhuma linha do programa sem ser executada.

Muito bem! Na próxima etapa, nesta nossa crescente evolução, você conhecerá a linguagem de programação que será o Java. É importante repetir que o mais importante para você é o conhecimento lógico. A linguagem de programação é importante, mas, sem o raciocínio lógico, os resultados não serão expressivos. Por isso, sempre recomendamos que as dúvidas sejam dirimidas e reforçamos que praticar é fundamental.

Um bom programador é que nem um bom piloto de avião: quanto mais horas de voo, melhor ele é!



3. Expressões e operadores

Podemos definir uma “Expressão” como um conjunto de operandos e operadores que geram um resultado.

Já vimos anteriormente algumas expressões simples, como:

Expressão	Operandos	Operadores	Resultado
valor * 2	valor, 2	*	Valor numérico
(nota1 + nota2) / 2	nota1, nota2, 2	+, /	Valor numérico
Media >= 6	Media, 6	>=	Valor lógico

Fonte: Elaborada pelos autores (2024).

Neste capítulo, veremos mais profundamente as expressões e os operadores, pois são elementos muito importantes e constantemente utilizados na programação.

As **expressões** são comumente conhecidas para a realização de cálculos, mas o que caracteriza uma expressão é o tipo de resultado gerado. Isso significa que podemos classificá-las como:

- Expressões aritméticas ou matemáticas.
- Expressões lógicas.
- Expressões literais ou strings.

Você possivelmente já conhece a expressão matemática e, talvez, a expressão lógica, devido ao conhecimento adquirido durante a sua vida acadêmica, mas vamos resgatar todas as informações necessárias, incluindo as regras e, naturalmente, seus operadores relacionados.

Uma dessas regras, e que é comum a todas, é a que trata da **prioridade de operadores**, pois teremos operadores que quando utilizados em uma expressão, terão determinada prioridade sobre outros. Para a alteração dessa prioridade, utilizam-se os parênteses. Um exemplo é a expressão que já utilizamos: $(nota1 + nota2) / 2$. Sem os parênteses, a divisão seria executada primeiro e, assim, teríamos, muito possivelmente, um resultado errado. Veremos detalhadamente as prioridades quando apresentarmos, na sequência, os tipos de expressões.

Adiante você irá conhecer mais um componente que pode fazer parte das expressões, que é a utilização ou a chamada de funções. De maneira mais simplificada, uma função pode ser considerada um comando que realiza procedimentos próprios e gera (ou retorna) um resultado específico. Mas não se preocupe com isso agora, pois só entenderemos e utilizaremos mais tarde, quando estivermos trabalhando com a nossa Linguagem de Programação.

3.1 Expressões aritméticas ou matemáticas

Expressões aritméticas ou matemáticas são aquelas em que o resultado da operação é um valor numérico. Nas expressões desse tipo, utilizamos operadores aritméticos e operandos que podem ser constantes e/ou variáveis numéricas.



Os operadores comuns utilizados nas expressões aritméticas ou matemáticas são:

- + adição
- subtração
- * multiplicação
- / divisão.

Mas atenção! Existe uma situação em que devemos tomar cuidado: a prioridade de operadores. Nesse tipo de expressão, a prioridade está estabelecida da seguinte forma:

01

* (multiplicação)
e
/ (divisão)

02

+ (adição)
e
- (subtração)

03

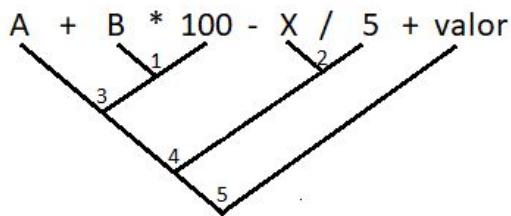
+ (adição)

04

- (subtração)

Perceba que a multiplicação e divisão têm o mesmo nível de prioridade, assim como a adição e subtração. Isso significa que quando houver mais de um operador de mesmo nível, a ordem de execução será da esquerda para a direita. Essa é uma característica de todas as expressões.

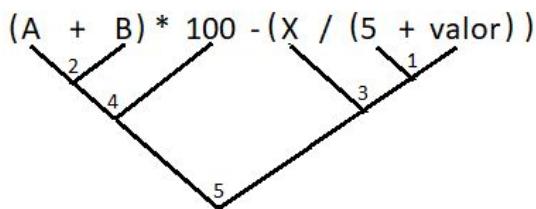
Veja o exemplo e atente-se à ordem das prioridades:



Fonte: Elaborada pelos autores (2024).

Quando necessário, para alterar a ordem natural dos operadores, devemos utilizar os parênteses: '()'.

Confira, agora, o mesmo exemplo anterior, mas alterando as prioridades com os parênteses:



Fonte: Elaborada pelos autores (2024).

A subexpressão 1 é a primeira a ser realizada, pois o nível dos parênteses também deve ser respeitado, por ser o nível mais interno. Na sequência, como temos duas subexpressões dentro de parênteses de mesmo nível, apesar de isso não afetar o resultado, seguimos a regra natural, que é da esquerda para a direita.

Expressões lógicas

Expressões lógicas são aquelas em que o resultado da operação é um valor lógico: verdadeiro ou falso. Nas expressões desse tipo, utilizamos operadores relacionais e/ou lógicos e operandos que podem ser constantes, variáveis e/ou outras expressões.

Os operadores relacionais utilizados apenas nas expressões lógicas são estes:

-  > maior
-  < menor
-  >= maior ou igual
-  <= menor ou igual
-  == igual
-  != diferente

Você talvez conheça o operador igual como '=' (apenas um), e o diferente como '≠'. Porém, utilizaremos a notação apresentada, pois é a utilizada na maior parte das linguagens de programação, assim como no Portugol Studio.

Os operadores lógicos que são utilizados apenas nas expressões lógicas são:

-  e conjunção
-  ou disjunção
-  nao negação

Uma expressão “e” é verdadeira se, e somente se, todas as condições forem verdadeiras. Uma expressão “ou” é verdadeira se pelo menos uma das condições for verdadeira. Uma expressão “não”, também conhecida como negação, inverte o valor da expressão ou condição apresentada, se ela for verdadeira inverte para falsa e vice-versa.

O quadro seguinte mostra todos os valores possíveis gerados pelos operadores lógicos: “e”, “ou” e “não”.

A	B	A e B	A ou B	Nao (A)
Verdadeiro	Verdadeiro	Verdadeiro	Verdadeiro	Falso
Verdadeiro	Falso	Falso	Verdadeiro	Falso
Falso	Verdadeiro	Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso	Falso	Verdadeiro

Fonte: Elaborado pelos autores (2024).

Quando começarmos a aprender alguma linguagem de programação, naturalmente, esses operadores lógicos deverão ser identificados de maneira diferente, como: `&&` (`e`), `||` (`ou`) e `!` (`não`) utilizados no Java, por exemplo.

As expressões lógicas serão muito utilizadas, principalmente, nas estruturas de controle (seleção e repetição).

Veja alguns exemplos:

		Considerando: X=5; Y=7; Resp='S' e Flag=falso:	
Expressão	Resultado	Expressão	Resultado
<code>7 >= 4</code>	Verdadeiro	<code>X > 5 e Y <= 7 e nao(Flag)</code>	Falso
<code>10 < 10</code>	Falso	<code>(Y - X) == 0 e (X >= Y) ou (X * 10) > 50</code>	Falso
<code>2+3 == 5</code>	Verdadeiro	<code>nao (X != Y) ou Resp == 'S'</code>	Verdadeiro
<code>'a' == 'A'</code>	Falso		

Fonte: Elaborada pelos autores (2024).

Nota: quando a comparação acontece entre dados caracter (texto), o código ASCII associado, que é uma representação numérica, é utilizado. No exemplo 'a' == 'A', o código ASCII de 'a' é o decimal 97, e o 'A' é o decimal 65. Portanto, são diferentes. Isso porque o computador é capaz de lidar apenas com números e no formato binário.

Veja a descrição de ASCII:



Código Padrão Americano para o Intercâmbio de Informação (do inglês, American Standard Code for Information Interchange) é um sistema de representação de letras, algarismos e sinais de pontuação e de controle, por meio de um sinal codificado em forma de código binário (cadeias de bits formada por vários 0 e 1). São 256 símbolos diferentes e cada um ocupa 1 byte (8 bits). (Wikipédia, 2024, on-line).



Quanto à prioridade de operadores, se houver, é a seguinte: primeiro os cálculos matemáticos; na sequência, os operadores relacionais e, em seguida e pela ordem, os operadores lógicos: 'nao', 'e' e 'ou'. Os parênteses não só alteram a prioridade, mas podem ser aproveitados para organizar a expressão.

Expressões literais

As expressões literais unem cadeias de caracteres em uma única cadeia de caracteres. Apesar de ser pouco conhecido, é o tipo de expressão mais simples dentre as expressões. Ela também é conhecida como expressão string, caracter ou alfanumérica, são expressões que lidam única e exclusivamente com cadeias de caracteres (ou textos) e/ou variáveis desse tipo.

O operador normalmente utilizado é o + (que parece o sinal de adição), que é chamado de **concatenação**.



Observe os seguintes exemplos:

"Gela" + "deira"

Nome + " " + Sobrenome

"Bom dia," + Nome + "!"

No primeiro exemplo, o resultado será “Geladeira”. Simples, pois se trata de dois literais.

Já no segundo exemplo, o resultado dependerá do que está armazenado nas variáveis Nome e Sobrenome. Perceba que, entre as variáveis, há um espaço em branco, justamente para garantir que haja um espaço entre elas. Vamos supor que Nome e Sobrenome tenham “Luiza” e “Moreira”, respectivamente. Assim, o resultado da expressão será “Luiza Moreira”.

Agora, no terceiro exemplo, não é tão difícil saber que o resultado será “Bom dia, Luiza！”, considerando que o conteúdo da variável nome seja “Luiza”.

Lembre-se de que a ordem da execução das concatenações, quando houver mais de uma, será, como já é de conhecimento, da esquerda para a direita. E, apesar de não ser comum, os parênteses são usados da mesma forma.

4. Introdução a Linguagem Java

Agora, vamos pôr em prática os conceitos aprendidos utilizando uma linguagem de programação, que nesse caso, será **Java**.

Java é uma linguagem de programação de alto nível e orientada a objetos, desenvolvida pela Sun Microsystems (agora parte da Oracle Corporation) na década de 1990 (Moura, 2024, on-line). É uma das linguagens mais utilizadas no mundo, especialmente em desenvolvimento de aplicativos corporativos, sistemas embarcados e aplicações para a web.

Para começar, é importante conhecer alguns conceitos básicos, como você pode conferir a seguir.



Orientação a objetos

Java é uma linguagem orientada a objetos, ou seja, tudo em Java é um objeto ou se baseia em um objeto. Isso promove a reutilização de código e a modularidade.



Portabilidade

Essa é uma das características mais marcantes do Java. Os programas Java são compilados em bytecode, que é executado na JVM (Java Virtual Machine). Isso significa que um programa Java compilado pode ser executado em qualquer dispositivo que tenha uma JVM instalada, tornando-o altamente portátil.



Sintaxe simples e familiar

A sintaxe do Java foi projetada para ser semelhante à de outras linguagens de programação, como C e C++, tornando mais fácil para os programadores que já estão familiarizados com essas linguagens aprenderem Java.



Compilação e execução

Para desenvolver programas Java, você escreve o código-fonte em arquivos com a extensão ".java". Assim, o código é compilado pelo compilador Java (javac) em bytecode, contido em arquivos com extensão ".class". Esses arquivos de bytecode podem ser executados em qualquer máquina virtual Java (JVM) usando o comando Java.



Estrutura do programa

Um programa Java é composto por uma ou mais classes. Cada classe contém variáveis e métodos. O método main() é o ponto de entrada para um programa Java, ou seja, é o ponto de início da execução do programa.



Gestão de memória

Java gerencia automaticamente a alocação e liberação de memória para objetos. Isso é feito por meio do mecanismo de coleta de lixo (garbage collection), que libera automaticamente a memória usada por objetos que não são mais referenciados pelo programa.

Cabe destacar que **bytecode** é o código de instruções de máquina gerado por compiladores para plataformas virtuais, como a JVM (Java Virtual Machine), no caso do Java.

Quando um programa Java é compilado, o código-fonte é traduzido para bytecode, que é então executado pela JVM. O bytecode é independente de plataforma, o que significa que pode ser executado em qualquer dispositivo que tenha uma implementação da JVM adequada. Isso proporciona a portabilidade do código Java, pois o mesmo bytecode pode ser executado em sistemas operacionais diferentes,

Bytecode: em termos simples, é um conjunto de instruções que são executadas pela máquina virtual.

desde que tenham uma JVM compatível.

Observe, a seguir, um exemplo de pseudocódigo que apresenta na tela a mensagem "Oi Mundo!".

</> Código na área

```
programa {
    funcao inicio ()
    {
        escreva("Oi Mundo\n")
    }
}
```

Agora, veja o código em Java para o pseudocódigo apresentado anteriormente.

</> Código na área

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

        System.out.print("Oi Mundo");
    }
}
```

A execução do código acima terá a seguinte saída: **Oi Mundo**. Vamos entender um pouco o que foi codificado no primeiro programa?



A **primeira linha** do arquivo declarou um nome de classe. Como ainda estudaremos o que é uma classe, em Orientação a Objetos, chamaremos, agora, de "programa". Então, a primeira linha declarou o programa com o nome "OiMundo". A abertura e o fechamento das chaves indicam um bloco de código. Portanto, tudo o que estiver lá dentro pertence ao programa OiMundo.

No bloco de código do programa, desenvolvemos um método chamado **main**. Por enquanto, iremos chamá-lo de procedimento ou função. Esse método serve para que o programa seja executado quando digitarmos o comando Java OiMundo. Isso significa que ele é o ponto de entrada do programa, então será executado automaticamente com a solicitação de execução do programa.

Cabe lembrar que o nome do método não pode ser alterado, pois, ainda que não pertença à sintaxe da linguagem, é um padrão do Java que significa um ponto inicial de um programa desenvolvido.

Além disso, o bloco de código delimitado pelas chaves deve ter uma ou mais linhas com a programação do que o sistema deve fazer. Observe:

</> Código na área

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

Em nosso exemplo, o programa apenas imprime "Oi mundo" na tela. Para fazer isso, usamos o método System.out.println.

Todo texto (string) em Java é delimitado por aspas duplas, e toda instrução (comando) deve terminar com um ponto e vírgula. Note também que o texto "Oi mundo" está entre parênteses, o que indica o início e término de um parâmetro do método:

</> Código na área

```
System.out.println("Oi mundo");
```

Acompanhe, a seguir, outros conceitos importantes da linguagem Java.

Comentários

Comentários são textos que podem ser incluídos no código-fonte. Geralmente, são utilizados para descrever como determinado programa ou bloco de código funciona.

Os comentários são ignorados pelo compilador, por isso, não modificam o comportamento do programa.

Em Java, você pode comentar blocos de códigos inteiros ou apenas uma linha. Para comentar uma única linha, faça como no exemplo a seguir:

</> Código na área

```
public class OiMundo {  
    public static void main(String[] args) {  
        // imprime uma mensagem na saída padrão  
        System.out.print("Oi Mundo");  
    }  
}
```

A execução do código acima terá a seguinte saída: **Oi Mundo.**

Para comentar um bloco de código, use `/* */` para abrir e fechar. Veja um exemplo:

</> Código na área

```
public class OiMundo {  
    public static void main(String[] args) {  
        /* esta linha será ignorada pelo compilador java  
         * System.out.print("esta instrucao sera ignorada  
         * tambem");  
         * e esta linha também será ignorada. */  
        System.out.print("Oi Mundo");  
    }  
}
```

Nós temos, também, alguns tipos de variáveis em Java, conforme será apresentado a seguir.

Variáveis em Java

Em Java, existem oito tipos de variáveis primitivas, que representam os tipos de dados básicos e fundamentais da linguagem. Conheça cada uma delas a seguir.

1.

byte



É um tipo inteiro de 8 bits com sinal, com um intervalo de -128 a 127. É frequentemente usado quando se trabalha com grandes volumes de dados de bytes, ou em situações em que o espaço é uma preocupação.

short

2.



É um tipo inteiro de 16 bits com sinal, com um intervalo de -32,768 a 32,767. Ele é útil quando o intervalo de valores esperado é maior do que o oferecido pelo tipo byte, mas ainda se deseja economizar espaço em comparação com um tipo int.

3.

int



É um tipo inteiro de 32 bits com sinal, com um intervalo de -2^{31} a $2^{31} - 1$. É o tipo de dados mais comum para representar números inteiros em Java.

long

4.



É um tipo inteiro de 64 bits com sinal, com um intervalo de -2^{63} a $2^{63} - 1$. É usado quando valores inteiros muito grandes são necessários.

5.

float



É um número de ponto flutuante de precisão simples de 32 bits, usado para representar números decimais. É importante observar que os números de ponto flutuante podem não representar com precisão números decimais exatos.

double

6.



É um número de ponto flutuante de precisão dupla de 64 bits. Ele oferece maior precisão do que o tipo float e é frequentemente usado para cálculos que requerem alta precisão.

7.

char

01011
11010
01011


É usado para representar caracteres únicos de 16 bits da tabela Unicode. Em Java, os caracteres são representados usando aspas simples, como 'A' ou 'a'.

boolean

É usado para representar valores booleanos, ou seja, verdadeiro ou falso.

Ele ocupa somente um bit de armazenamento e pode ter apenas dois valores possíveis: true ou false.

8.



Esses tipos de variáveis primitivas fornecem os blocos de construção básicos para a manipulação de dados em Java, por isso, é importante entender seus comportamentos e suas limitações ao escrever código Java. Veja, adiante, como declarar essas variáveis.

Como declarar variáveis em Java

Em Java, é necessário declarar as variáveis com um tipo fixo, para que possam ser usadas. Por exemplo, uma variável do tipo inteiro não poderá ser alterada para um tipo real (decimal). Para que você possa compreender como funciona na prática, vamos declarar uma variável chamada **quantidade**, do tipo int, que é capaz de armazenar apenas valores inteiros negativos ou positivos. Observe:

</> Código na área

```
int quantidade; // declarando variável inteira
```

Desse modo, apenas declaramos a variável acima, isto é, não atribuímos nenhum valor a ela. E como atribuir valores? Confira a seguir!

Como atribuir valores a variáveis

Para atribuir valores, devemos utilizar o operador = (igual) seguido por um número inteiro. Por exemplo:

quantidade = 10; // atribuindo o valor 10

Nesse exemplo, a variável quantidade tem o valor 10. Se for preciso alterar o valor da variável quantidade, podemos atribuir um novo valor a ela. Vamos considerar, por exemplo, que a variável deve ter o valor 15:

quantidade = 15; // atribuindo o valor 15

Frequentemente, precisamos mostrar o valor de uma variável na tela do usuário. Podemos fazer isso de uma forma bem simples, utilizando System.out.println, passando como parâmetro o nome da variável.

</> Código na área

```
System.out.println(quantidade);
```

Acompanhe, a seguir, um exemplo incluindo a impressão da variável quantidade entre a atribuição do valor 10 e do valor 15 para podermos ver o valor das variáveis antes e depois de ser modificado:

</> Código na área

```
public class Variaveis {  
    public static void main(String[] args) {  
        int quantidade;  
        quantidade = 10;  
        System.out.println(quantidade);  
        quantidade = 15;  
        System.out.println(quantidade);  
    }  
}
```

A execução desse código terá a seguinte saída:

10

15

A linguagem Java é *case sensitive*, ou seja, ela distingue letras maiúsculas de letras minúsculas. Portanto, muito cuidado!

As regras para nome de variáveis que você aprenderá em pseudocódigo continuam valendo, mas lembre-se de que Java tem um conjunto de palavras reservadas que você aprenderá com o tempo. Essas palavras não podem ser utilizadas para nome de variável.

Agora que você já dominou o conceito sobre as variáveis em Java, abordaremos os operadores aritméticos. Vamos lá?

Operadores Aritméticos

Existem cinco operadores aritméticos em Java que podemos usar para fazer cálculos matemáticos. Precisamos considerar que temos operações de adição (+), subtração (-), multiplicação (*), divisão (/) ou módulo (%). Lembre-se de que o módulo é o resto da divisão entre dois números. A seguir, observe um exemplo utilizando esses operadores:

</> Código na área

```
public class OperadoresAritmeticos{
    public static void main(String[] args) {
        int soma, subtracao, multiplicacao, divisao, resto;
        soma = 2 + 10;
        subtracao = 2 - 10;
        multiplicacao = 2 * 10;
        divisao = 10 / 2;
        resto = 10 % 3;
        System.out.println(soma);
        System.out.println(subtracao);
        System.out.println(multiplicacao);
        System.out.println(divisao);
        System.out.println(resto);
    }
}
```

A execução do código acima terá a seguinte saída:

12
-8
20
5
1

Operadores Relacionais

Em Java, os operadores relacionais são usados para comparar dois valores e determinar a relação entre eles. Eles retornam um valor booleano (true ou false) que indica se a comparação é verdadeira ou falsa. Na tabela a seguir, estão os operadores relacionais em Java:

Operador	Sinal	Descrição	Exemplo
Igual a	<code>==</code>	Verifica se os dois operandos são iguais.	<code>op1 == op2</code>
Diferente de	<code>!=</code>	Verifica se os dois operandos são diferentes.	<code>op1 != op2</code>
Maior que	<code>></code>	Verifica se o operando da esquerda é maior que o operando da direita.	<code>op1 > op2</code>
Menor que	<code><</code>	Verifica se o operando da esquerda é menor que o operando da direita.	<code>op1 < op2</code>
Maior ou igual a	<code>>=</code>	Verifica se o operando da esquerda é maior ou igual ao operando da direita.	<code>op1 >= op2</code>
Menor ou igual a	<code><=</code>	Verifica se o operando da esquerda é menor ou igual ao operando da direita.	<code>op1 <= op2</code>

Fonte: Elaborada pelos autores (2024).

Em um programa Java, os operadores relacionais descritos na tabela anterior podem ser testados com o código seguinte:

</> Código na área

```
public class OperadoresRelacionaisExemplo {  
    public static void main(String[] args) {  
        // Operadores relacionais  
        int x = 5;  
        int y = 10;  
  
        System.out.println("Operadores relacionais:");  
        System.out.println("x == y: " + (x == y));      // false  
        System.out.println("x != y: " + (x != y));      // true  
        System.out.println("x > y: " + (x > y));      // false  
        System.out.println("x < y: " + (x < y));      // true  
        System.out.println("x >= y: " + (x >= y));     // false  
        System.out.println("x <= y: " + (x <= y));     // true  
    }  
}
```

Esses operadores são frequentemente usados em estruturas de controle de fluxo, como condicionais, repetições, e em expressões de atribuição condicional, as quais serão exemplificadas nos itens 5 e 6.

Operadores Lógicos

Em Java, os operadores lógicos são utilizados para combinar expressões booleanas e realizar operações lógicas sobre elas. Existem três operadores lógicos principais em Java:

Operador	Sinal	Descrição	Exemplo
E (and) lógico	&&	Retorna true se tanto a expressão1 quanto a expressão2 forem true, caso contrário, retorna false.	exp1 && exp2
OU (or) lógico		Retorna true se pelo menos uma das expressões (expressão1 ou expressão2) for true, caso contrário, retorna false.	op1 >= op2
NEGAÇÃO (not)	!	Retorna true se a expressão for false, e retorna false se a expressão for true.	!expressão

Fonte: Elaborada pelos autores (2024).

Em um programa Java, os operadores lógicos descritos na tabela anterior podem ser testados com o código seguinte:

</> Código na área

```
public class OperadoresLogicosExemplo {
    public static void main(String[] args) {
        // Operadores lógicos
        boolean a = true;
        boolean b = false;

        System.out.println("\nOperadores lógicos:");
        System.out.println("a && b: " + (a && b));      // false
        System.out.println("a || b: " + (a || b));      // true
        System.out.println("!a: " + (!a));            // false
    }
}
```

Esses operadores também serão usados em estruturas de controle de fluxo, como condicionais e repetições, e em expressões de atribuição condicional, as quais serão exemplificadas nos itens “Estruturas de Controle de Seleção” e “Estruturas de

Controle de Repetição”.

Na próxima etapa, você vai conhecer como funciona a classe em Java chamada Strings. Vamos lá?

Trabalhando com Strings

Em Java, a classe String é uma classe fundamental que representa uma sequência de caracteres. Ela é parte da biblioteca padrão do Java e é frequentemente utilizada em programas Java para manipulação de texto. A classe String em Java é imutável, o que significa que uma vez que uma instância de String é criada, seu conteúdo não pode ser alterado. Isso implica que qualquer operação que pareça modificar uma String, na verdade, cria uma nova String com o conteúdo modificado, deixando a instância original intacta.

A imutabilidade das Strings em Java tem implicações na eficiência e no gerenciamento de memória, e é importante entender isso ao trabalhar com elas.



Para apresentar um texto na tela usando Java, tudo o que você precisa fazer é colocá-lo entre aspas duplas dentro de um System.out.println.

Se for necessário juntar um texto com o valor de uma variável, é possível concatená-los. Na linguagem Java, usamos o símbolo + (mais) para fazer isso.

</> Código na área

```
public class OperadoresAritmeticos{  
    public static void main(String[] args) {  
  
        int soma, subtracao, multiplicacao, divisao, resto;  
        soma = 2 + 10;  
        subtracao = 2 - 10;  
        multiplicacao = 2 * 10;  
        divisao = 10 / 2;  
        resto = 10 % 3;  
        System.out.println("Resultado da soma: " + soma);  
        System.out.println("Resultado da subtracao: " +  
        subtracao);  
        System.out.println("Resultado da multiplicacao: " +  
        multiplicacao);  
        System.out.println("Resultado da divisao: " + divisao);  
        System.out.println("Resultado do resto: " + resto);  
    }  
}
```

A execução desse código terá a seguinte saída:



Resultado da soma: 12
Resultado da subtração: -8
Resultado da multiplicação: 20
Resultado da divisão: 5
Resultado do resto: 1

No exemplo a seguir, declaramos a variável com nome do tipo String e a idade do tipo int, e depois imprimimos na tela uma mensagem, concatenando o nome e a idade mais outras informações (textos):

</> Código na área

```
public class ExemploString{  
    public static void main(String[] args) {  
        int idade = 50;  
        String nome = "Marcos";  
        System.out.println(nome + " tem " + idade + " anos ");  
    }  
}
```

A execução desse código terá a seguinte saída: **Marcos tem 50 anos.**

Veja, adiante, como solicitar informações ao usuário na execução de um programa.

Recebendo entrada de dados

Até este ponto, demonstramos exemplos empregando variáveis com valores diretamente inseridos no código-fonte (conhecidos como literais), sem envolvimento de entrada de dados pelo usuário. Agora, é hora de elevar o nível de nossos códigos e compreender como requisitar informações ao usuário durante a execução do programa.

A maneira mais simples de realizar isso em Java é por meio de uma classe denominada Scanner. Por ora, não é necessário compreender profundamente o conceito de classe, nem entender o funcionamento específico do Scanner.

</> Código na área

```
import java.util.Scanner;

public class EntradaDeDados{
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);
        System.out.print("Nome do produto: ");
        String nome = leia.nextLine();
        System.out.print("Quantidade de produto no estoque: ");
        int qtd = leia.nextInt();

        System.out.println("**** Dados do produto ****");
        System.out.println("Produto.....: " + nome);
        System.out.println("Quantidade...: " + qtd);
    }
}
```

A execução desse código terá a seguinte saída:

</> Código na área

```
Nome do produto: Tênis
Quantidade de produto no estoque: 10

**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 10
```

Para usar a classe Scanner, é necessário importá-la em nosso código-fonte. Para fazer isso, basta incluir a linha seguinte no topo do arquivo do código-fonte:

</> Código na área

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

Declaramos uma variável nomeada como leia do tipo Scanner. É por meio dessa variável que serão feitas as chamadas para entrada (via teclado) de dados do tipo inteiro, real, string:

</> Código na área

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

Finalmente, para solicitar a entrada de um texto, invocamos o método `nextLine()` por meio da variável entrada:

</> Código na área

```
String nome = leia.nextLine();
```

Neste momento, o programa ficará paralisado aguardando o usuário digitar alguma coisa. Quando a tecla Enter for pressionada, o conteúdo informado será atribuído à variável nome.

A classe Scanner tem vários métodos para obter dados já convertidos em tipos específicos. Depois de fazer a leitura do nome, foi solicitada a quantidade de produtos, usamos o método para obter entrada dados do tipo int. O programa solicitará, além do nome do produto, sua quantidade:

</> Código na área

```
qtd = leia.nextInt();
```



Deu bug?

Podemos chamar o método nextLine() depois de ter chamado nextInt() ou nextDouble() por meio da mesma variável do Scanner?

Neste caso, não tente chamar o método nextLine() depois de ter chamado nextInt() ou nextDouble() através da mesma variável do Scanner, pois isso não funciona bem, haja vista que fazer a entrada de um inteiro ou double com nextInt ou nextdouble, implica no comportamento do método **nextLine()**, que lê toda a linha de entrada (número), incluindo o caractere de nova linha ("\n" - ao pressionar a tecla enter), mas não consome esse caractere. Como resultado, quando você tenta ler uma linha após nextInt ou nextFloat com um **nextLine()**, ele imediatamente retorna uma string vazia, pois o caractere de nova linha ainda está pendente no buffer de entrada. Uma forma de resolver é criando duas variáveis do tipo Scanner para solucionar esse problema, uma para ler valores numéricos e outra para ler strings. Verifique o comportamento do código a seguir:

</> Código na área

```
import java.util.Scanner;

public class EntradaDeDados{
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);

        System.out.print("Quantidade de produto no estoque: ");
        int qtd = leia.nextInt();
        System.out.print("Nome do produto: ");
        String nome = leia.nextLine();

        System.out.println("**** Dados do produto ****");
        System.out.println("Produto.....: " + nome);
        System.out.println("Quantidade...: " + qtd);
    }
}
```

Outra forma de resolver o problema é usar `nextLine()` para fazer todas as entradas. Neste caso, quando o valor for numérico, é necessário fazer a conversão de `String` para o tipo numérico equivalente. Veja o exemplo a seguir:

</> Código na área

```
import java.util.Scanner;

public class EntradaDeDadosComNextLine {

    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);
        String strEntrada;
```

```
        System.out.print("Quantidade de produto no  
Estoque...: ");  
        strEntrada = leia.nextLine();  
        //a linha abaixo faz o parse de String para int  
        int qtd = Integer.parseInt(strEntrada);  
        System.out.print("Nome do  
produto.....: ");  
        //a linha abaixo, como é uma string, não requer  
        conversão  
        String nome = leia.nextLine();  
        System.out.print("Preço do produto.....: ");  
        strEntrada = leia.nextLine();  
        //faz o parse de String para float  
        float preco = Float.parseFloat(strEntrada);  
  
        System.out.println("**** Dados do produto ****");  
        System.out.println("Produto.....: " + nome);  
        System.out.println("Quantidade...: " + qtd);  
        System.out.println("Preço.....: " + preco);  
    }  
}
```

Para conhecer a história do surgimento do Java, recomendamos acessar o conteúdo seguinte:



A seguir, você poderá conhecer as Estruturas de Controle de Seleção. Confira!

5. Estruturas de Controle de Seleção

As estruturas de decisão são elementos fundamentais em linguagens de programação que permitem que um programa tome decisões com base em determinadas condições. Portanto, elas permitem que o fluxo de execução do programa seja alterado com base em valores fornecidos durante a execução do programa.

Além disso, essas estruturas servem para controlar o fluxo de execução do programa, permitindo que ele execute diferentes blocos de código, dependendo de condições específicas. Isso é essencial para que programas possam ser mais dinâmicos e reativos às diferentes situações que podem ocorrer durante sua execução.

Cabe ressaltar, também, que as estruturas de decisão são aplicadas sempre que um programa precisa tomar decisões com base em certas condições. Alguns cenários comuns em que as estruturas de decisão são aplicadas incluem:

Validação de Entrada de Dados

Verificar se os dados fornecidos pelo usuário atendem a certos critérios antes de serem processados.



Tratamento de Exceções

Decidir como lidar com erros ou exceções que ocorrem durante a execução do programa.



Roteamento de Fluxo de Execução

Determinar qual bloco de código deve ser executado com base em diferentes condições.



Personalização de Experiência do Usuário

Adaptar o comportamento do programa com base nas preferências ou ações do usuário.



Implementação de Lógica de Negócios

Realizar operações específicas com base em critérios lógicos definidos.

Em resumo, as estruturas de decisão são aplicadas em praticamente todos os programas para permitir que eles tomem decisões dinâmicas com base em diferentes condições durante a execução. Elas são fundamentais para a criação de programas flexíveis e adaptáveis.

A seguir, vamos abordar as principais estruturas de controle de decisão, sendo elas, **se (if)** e **senão (else)** simples e composto, **escolha caso (switch case)** e **condicional ternário**.

Estrutura de controle SE e SENÃO

A estrutura de controle de decisão SE (if) e SENÃO (else) é fundamental na programação para direcionar o fluxo de execução do programa com base em condições específicas. Uma instrução SE permite que o programa avalie uma condição e execute um bloco de código caso ela seja verdadeira. Por exemplo, ao verificar SE uma variável é maior que um determinado valor, o programa pode executar uma determinada ação apenas se essa condição for atendida. Já o SENÃO é utilizado em conjunto com o SE para especificar um bloco de código a ser executado caso a condição do SE seja falsa. Deste modo, o programa pode seguir um curso alternativo de ação, caso a condição inicial não seja satisfeita.



Além disso, a estrutura SE pode ser composta por vários SENÃO SE (*else if*), permitindo avaliar múltiplas condições encadeadas de forma sequencial, o que amplia as possibilidades de tomada de decisão em um programa. Essas estruturas são essenciais para a lógica de programação e são amplamente utilizadas em diversas aplicações para controlar o comportamento do programa de acordo com as circunstâncias. Nas subseções a seguir, são apresentadas as formas como utilizar as instruções SE e SENÃO no seu programa.

SE (IF)

A instrução **SE (if)** serve para controlar um fluxo básico de programa e executar uma sequência de instruções específicas caso uma determinada condição seja verdadeira. Quando implementada para verificar uma única possibilidade, ela é chamada de **simples**, e sua estrutura é definida da seguinte forma:

</> Código na área

```
se (condição) {  
    // Executa quando a condição for verdadeira  
    instrução_1  
    instrução_2  
    instrução_n  
}
```

A condição será sempre fornecida por meio de uma variável booleana ou por meio de uma expressão condicional, de forma que o valor verificado sempre seja ser **VERDADEIRO** ou **FALSO**. No entanto, a sequência de instruções só será executada caso a condição seja verdadeira.

Na linguagem de programação Java, quando uma única instrução deve ser executada caso a condição seja verdadeira, a sintaxe é da seguinte forma:

</> Código na área

```
if (condição)  
    instrução;
```

Caso tenha mais de uma instrução a ser executada quando a condição for verdadeira, deve-se envolver as instruções em um bloco iniciando com o caractere abre chaves (`{`) e finalizando com fecha chaves (`}`). A sintaxe deve seguir esta estrutura:

</> Código na área

```
if (condição) {  
    instrução_1;  
    instrução_2;  
    instrução_n;  
}
```

Imagine, por exemplo, um algoritmo em que determinado estudante somente estará aprovado se sua média for maior ou igual a 7 (uso do operador relacional **maior ou igual** “`>=`”).

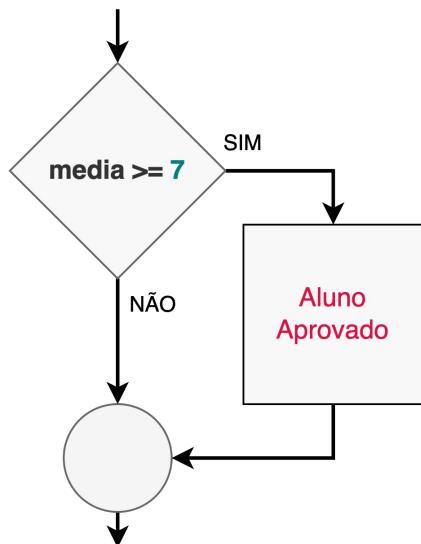
Observe como fica em pseudocódigo:

</> Código na área

```
se (media >= 7) {  
    escreva("Aluno Aprovado")  
}
```

Em diagrama de bloco, fica da seguinte maneira:

Diagrama de bloco 1



Fonte: Elaborado pelos autores (2024).

Já em código Java, fica da seguinte maneira:

</> Código na área

```
if (media >= 7) {  
    System.out.println("Aluno Aprovado");  
}
```

Confira, a seguir, um exemplo do uso da instrução SE em um programa Java para avaliar o valor da variável **qtd**. A instrução da linha 16 somente será executada se a condição **SE** for verdadeira, isto é, se **qtd <= 0** (uso do operador relacional **menor ou igual “<=”**).

</> Código na área

```
import java.util.Scanner;

public class InstrucaoIF {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);
        System.out.print("Nome do produto: ");
        String nome = leia.nextLine();
        System.out.print("Quantidade de produto no estoque: ");
        int qtd = leia.nextInt();

        System.out.println("**** Dados do produto ****");
        System.out.println("Produto.....: " + nome);
        System.out.println("Quantidade...: " + qtd);

        if (qtd <= 0) {
            System.out.println("Produto sem estoque");
        }
    }
}
```

A execução do código acima terá a seguinte saída caso a quantidade de estoque informada seja maior que zero:

</> Código na área

```
Nome do produto: Tênis
Quantidade de produto no estoque: 10
**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 10
```

Ao contrário, se a quantidade ≤ 0 , então será exibida a mensagem “**Produto sem estoque**”. A seguir, o resultado após a execução do programa:

</> Código na área

```
Nome do produto: Tênis
Quantidade de produto no estoque: 0
**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 0
Produto sem estoque
```

Como já sabemos, toda instrução Java deve terminar com ponto e vírgula “;”. Qual será o resultado emitido se você colocar “;” no final da instrução if, como no exemplo a seguir?

</> Código na área

```
if (qtd <= 0); {
    System.out.println("Produto sem estoque");
}
```

Não tenha dúvidas, implemente esse código no seu programa e verifique as saídas.

SE .. SENÃO (IF .. ELSE)

A estrutura **SE .. SENÃO**, também conhecida como SE composto, é utilizada quando há uma alternativa a ser executada quando a condição **SE** não for satisfeita.

Ela funciona de forma excludente e complementar, ou seja, sempre uma sequência de instruções será executada. Caso a condição seja verdadeira, as instruções no bloco **SE** serão executadas. Do contrário, as instruções no bloco **SENÃO** serão executadas.

</> Código na área

```
se(condição) {  
    // Executa quando a condição for verdadeira  
    instrução_1  
    instrução_2  
    instrução_n  
} senao {  
    // Executa quando a condição for falsa  
    instrução_1  
    instrução_2  
    instrução_n  
}
```

Na linguagem de programação Java, a sintaxe desta instrução é:

</> Código na área

```
if (condição)  
    instrução;  
else  
    instrução;
```

Caso tenha mais de uma instrução a ser executada se a condição for verdadeira ou falsa, a sintaxe deve seguir esta estrutura:

</> Código na área

```
if (condição) {  
    instrução_1;  
    instrução_2;  
    instrução_n;  
} else {  
    instrução_1;  
    instrução_2;  
    instrução_n;  
}
```

Imagine um algoritmo que determinado aluno somente estará aprovado se sua média for maior ou igual a 7, do contrário, ele será reprovado.

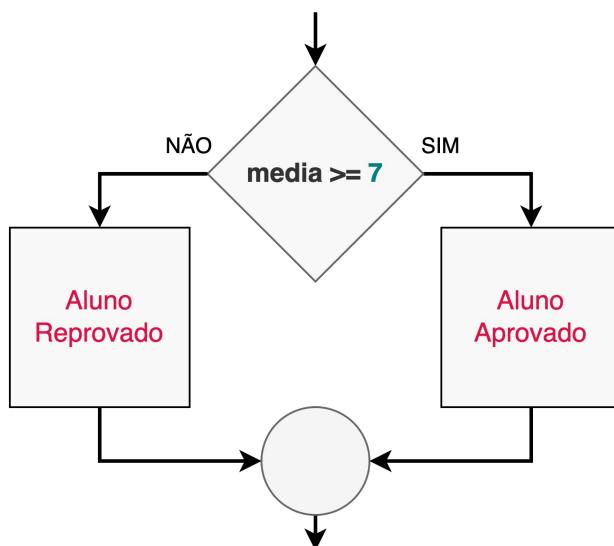
Veja como fica em pseudocódigo:

</> Código na área

```
se (media >= 7) {  
    escreva("Aluno Aprovado")  
} senao {  
    escreva(Aluno Reprovado)  
}
```

Em diagrama de bloco, temos o seguinte:

Diagrama de bloco 2



Fonte: Elaborado pelos autores (2024).

Em código Java, é implementado da seguinte maneira:

</> Código na área

```
if (media >= 7) {  
    System.out.println("Aluno Aprovado");  
} else {  
    System.out.println("Aluno Reprovado");  
}
```

A seguir, há um exemplo do uso da instrução **SE** e **SENAO** em um programa Java. **Se** a **qtd > 0**, será exibida na tela a mensagem "**Produto disponível no estoque**", caso contrário, "**Produto indisponível no estoque**".

</> Código na área

```
public class InstrucaoIfElse {  
    public static void main(String[] args) {  
        Scanner leia = new Scanner(System.in);  
        System.out.print("Nome do produto: ");  
        String nome = leia.nextLine();  
        System.out.print("Quantidade de produto no estoque: ");  
        int qtd = leia.nextInt();  
  
        System.out.println("**** Dados do produto ****");  
        System.out.println("Produto.....: " + nome);  
        System.out.println("Quantidade...: " + qtd);  
  
        if (qtd > 0) {  
            System.out.println("Produto disponível no  
estoque");  
        } else {  
            System.out.println("Produto indisponível no  
estoque");  
        }  
    }  
}
```

A execução desse código terá a seguinte saída caso a quantidade de estoque informada seja maior que zero:

</> Código na área

```
Nome do produto: Tênis  
Quantidade de produto no estoque: 10  
**** Dados do produto ****  
Produto.....: Tênis  
Quantidade...: 10  
Produto disponível no estoque
```

Ao contrário, se a quantidade ≤ 0 , então, será exibida a mensagem “Produto indisponível no estoque”. A figura a seguir mostra essa situação:

</> Código na área

```
Nome do produto: Tênis
Quantidade de produto no estoque: 0
**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 0
Produto indisponível no estoque
```

Agora imagine um algoritmo em que determinado aluno somente estará aprovado se sua média for maior ou igual a 7, ficará em recuperação se a média for menor que 7 e maior ou igual a 5 e reprovado se a média for menor que 5.

Utilizando os comandos **SE .. SENÃO**, podemos criar a seguinte solução:

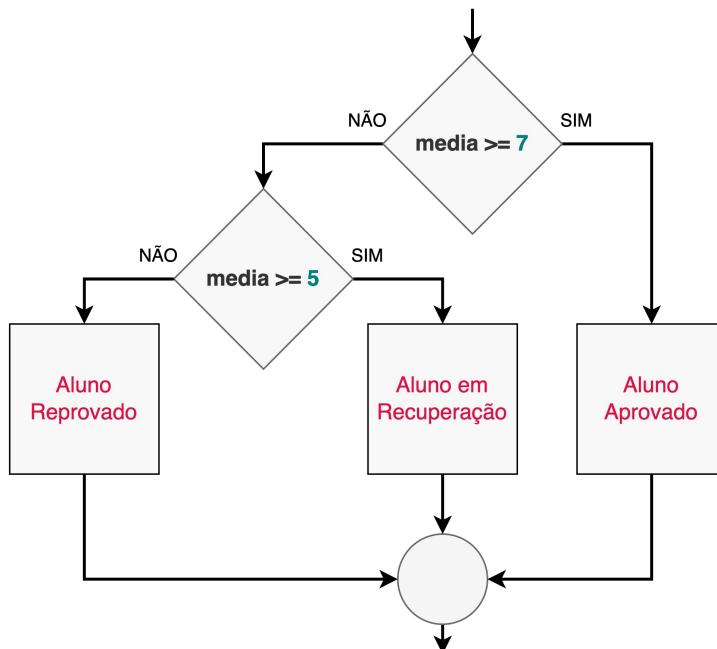
Veja como fica em pseudocódigo:

</> Código na área

```
se(media >= 7) {
    // Executa quando a media for maior ou igual a 7
    escreva("Aluno Aprovado")
} senao {
    se(media >= 5) {
        /* Executa quando a media, não sendo maior ou
        igual a 7,
        for maior ou igual a 5 */
        escreva("Aluno em Recuperação")
    } senao {
        /* Executa quando a media, não sendo maior ou
        igual a 7,
        for menor que 5 */
        escreva("Aluno Reprovado")
    }
}
```

Em diagrama de bloco, fica assim:

Diagrama de bloco 3



Fonte: Elaborado pelos autores (2024).

Em código Java, a implementação fica assim:

</> Código na área

```

if (media >= 7) {
    // Executa quando a media for maior ou igual a 7
    System.out.println("Aluno Aprovado");
} else {
    if (media >= 5) {
        /* Executa quando a media, não sendo maior ou
        igual a 7,
        for maior ou igual a 5 */
        System.out.println("Aluno em Recuperação");
    } else {
        /* Executa quando a media, não sendo maior ou
        igual a 7,
        for menor que 5 */
        System.out.println("Aluno Reprovado");
    }
}
  
```

E se existir mais de uma condição a ser testada? Vamos aprender a seguir.

SE .. SENÃO SE (IF .. ELSE IF - IF Aninhado)

A estrutura **SE .. SENÃO SE** (também conhecida como **IF Aninhado**) é utilizada quando há mais de uma condição a ser testada.

Elas testarão cada condição até que uma delas seja satisfeita. Caso uma condição seja verdadeira, as instruções daquele bloco serão executadas e o programa não testará as demais condições.

</> Código na área

```
se(condição_1) {  
    // Executa quando a condição 1 for verdadeira  
    instrução_1  
    instrução_2  
    instrução_n  
} senao (condição_2) {  
    // Executa quando a condição 2 for verdadeira  
    instrução_1  
    instrução_2  
    instrução_n  
} senao {  
    // Executa quando todas as condições forem falsas  
    instrução_1  
    instrução_2  
    instrução_n  
}
```

Agora acompanhe o exemplo da média do aluno com IF Aninhado, mantendo as mesmas condições: aprovado se sua média for maior ou igual a 7, em recuperação se a média for menor que 7 e maior ou igual a 5 e reprovado se a média for menor que 5.

Observe em pseudocódigo:

</> Código na área

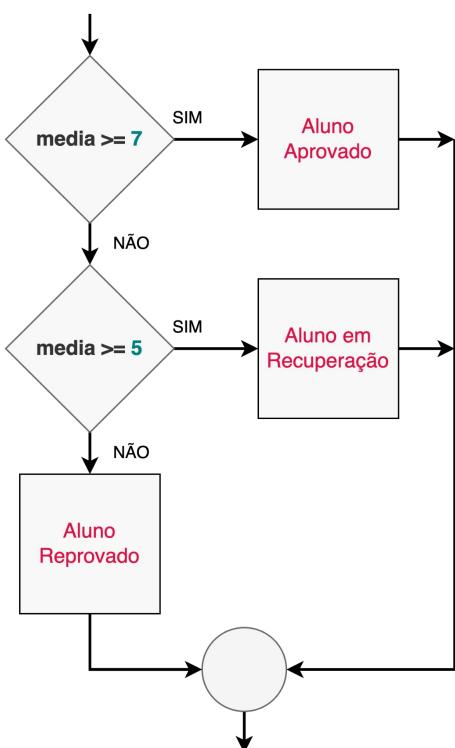
```

se(media >= 7) {
    // Executa quando a media for maior ou igual a 7
    escreva("Aluno Aprovado")
} senao (media >= 5) {
    /* Executa quando a media, não sendo maior ou igual
    a 7,
    for maior ou igual a 5 */
    escreva ("Aluno em Recuperação")
} senao {
    /* Executa quando a media for menor que 5 */
    escreva ("Aluno Reprovado")
}

```

Em diagrama de bloco, a solução fica da seguinte forma:

Diagrama de bloco 4



Fonte: Elaborado pelos autores (2024).

Agora, em código Java, a implementação é realizada assim:

</> Código na área

```
if (media >= 7) {  
    // Executa quando a media for maior ou igual a 7  
    System.out.println("Aluno Aprovado");  
} else if (media >= 5) {  
    /* Executa quando a media, não sendo maior ou igual  
    a 7,  
    for maior ou igual a 5 */  
    System.out.println("Aluno em Recuperação");  
}  
else {  
    /* Executa quando a media for menor que 5 */  
    System.out.println("Aluno Reprovado");  
}
```

Acompanhe, a seguir, um exemplo do uso da instrução **SE** e **SENAO SE** em um programa Java. No exemplo, a variável **qtd** pode passar por até três condições (verificações), se uma delas for satisfeita, então a respectiva instrução é executada, caso contrário, será exibida a mensagem: "O estoque do produto está dentro dos limites desejáveis.". Nesse mesmo exemplo, é possível observar o uso dos operadores relacional igualdade “==”, maior que “>” e menor que “<”.

</> Código na área

```
import java.util.Scanner;

public class InstrucaoIfElseIf {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);
        System.out.println(">>> Entre com os dados do Produto
<<<");

        System.out.print("Nome.....: ");
        String nome = leia.nextLine();
        System.out.print("Quantidade atual...: ");
        int qtd = leia.nextInt();
        System.out.print("Quantidade máxima..: ");
        int qtdMaxima = leia.nextInt();
        System.out.print("Quantidade mínima..: ");
        int qtdMinima = leia.nextInt();

        System.out.println("**** Dados do produto ****");
        System.out.println("Produto.....: " + nome);
        System.out.println("Quantidade....: " + qtd);
        System.out.println("Qtd. Máxima...: " + qtdMaxima);
        System.out.println("Qtd. Mínima..: " + qtdMinima);

        if (qtd == 0) {
            System.out.println("Produto indisponível no
estoque!");
        } else if (qtd > qtdMaxima) {
            System.out.println("A quantidade de produtos está
acima do "
                    + "limite máximo desejável!");
        } else if (qtd < qtdMinima) {
            System.out.println("Este produto está com estoque
baixo. "
                    + "\nEfetue a compra antes que
acabe!");
        } else {
            System.out.println("O estoque do produto está
dentro dos limites desejáveis.");
        }
    }
}
```

A execução do código apresentado terá a seguinte saída caso a quantidade de estoque informada seja zero.

</> Código na área

```
>>> Entre com os dados do Produto <<<
Nome.....: Tênis
Quantidade atual...: 0
Quantidade máxima...: 100
Quantidade mínima...: 10
**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 0
Qtd. Máxima...: 100
Qtd. Mínima...: 10
Produto indisponível no estoque!
```

Se **qtd > qtdMaxima** (uso do operador relacional **maior que ">"**), então, teremos a seguinte saída:

</> Código na área

```
>>> Entre com os dados do Produto <<<
Nome.....: Tênis
Quantidade atual...: 110
Quantidade máxima...: 100
Quantidade mínima...: 10
**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 110
Qtd. Máxima...: 100
Qtd. Mínima...: 10
A quantidade de produtos está acima do limite máximo desejável!
```

Se **qtd < qtdMinima** (uso do operador relacional **menor que “<”**), então, a saída será a seguinte:

</> Código na área

```
>>> Entre com os dados do Produto <<<
Nome.....: Tênis
Quantidade atual...: 5
Quantidade máxima...: 100
Quantidade mínima...: 10
**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 5
Qtd. Máxima...: 100
Qtd. Mínima...: 10
Este produto está com estoque baixo.
Efetue a compra antes que acabe!
```

Por fim, caso a **quantidade (qtd)** inserida estiver dentro dos limites desejáveis, então, o programa executará a saída mostrada a seguir:

</> Código na área

```
>>> Entre com os dados do Produto <<<
Nome.....: Tênis
Quantidade atual...: 50
Quantidade máxima...: 100
Quantidade mínima...: 10
**** Dados do produto ****
Produto.....: Tênis
Quantidade...: 50
Qtd. Máxima...: 100
Qtd. Mínima...: 10
O estoque do produto está dentro dos limites desejáveis.
```

Para fins de demonstração, o código seguinte apresenta exemplos da aplicação de **operadores relacionais** e **lógicos** em expressões de controle condicionais em um programa Java:

</> Código na área

```
public class OperadoresExemplo {  
    public static void main(String[] args) {  
        // Operadores relacionais  
        int x = 5;  
        int y = 10;  
  
        System.out.println("Operadores relacionais:");  
        if (x == y) {  
            System.out.println("x é igual a y.");  
        } else {  
            System.out.println("x não é igual a y.");  
        }  
  
        if (x != y) {  
            System.out.println("x é diferente de y.");  
        } else {  
            System.out.println("x é igual a y.");  
        }  
  
        if (x > y) {  
            System.out.println("x é maior que y.");  
        } else {  
            System.out.println("x não é maior que y.");  
        }  
  
        if (x < y) {  
            System.out.println("x é menor que y.");  
        } else {  
            System.out.println("x não é menor que y.");  
        }  
  
        if (x >= y) {  
            System.out.println("x é maior ou igual a y.");  
        } else {  
            System.out.println("x não é maior ou igual a y.");  
        }  
        if (x <= y) {  
            System.out.println("x é menor ou igual a y.");  
        } else {  
            System.out.println("x não é menor ou igual a y.");  
        }  
    }  
}
```

```
// Operadores lógicos
boolean a = true;
boolean b = false;

System.out.println("\nOperadores lógicos:");
if (a && b) {
    System.out.println("a e b são verdadeiros.");
} else {
    System.out.println("a e b não são ambos
verdadeiros.");
}

if (a || b) {
    System.out.println("a ou b é verdadeiro.");
} else {
    System.out.println("a e b são ambos falsos.");
}

if (!a) {
    System.out.println("a é falso.");
} else {
    System.out.println("a é verdadeiro.");
}
}
```

Até aqui, você conheceu as diversas condições da estrutura **SE .. SENÃO (IF .. ELSE)**. A partir de agora, você vai aprender sobre uma estrutura semelhante e simplificada.

Operador condicional ternário

O Condisional ternário ou operador ternário (IF ternário) é uma forma reduzida e simplificada semelhante ao **IF .. ELSE**, sendo escrita em apenas uma linha.

</> Código na área

```
(Condição) ? instrução_1 : instrução_2;
```

Caso a condição seja verdadeira, a instrução após o sinal de **?** será executada (**instrução_1**). Do contrário, a instrução após o sinal **:** é que será executada (**instrução_2**).

Veja como fica o exemplo da média do aluno em pseudocódigo:

</> Código na área

```
(media >= 7) ? ESCREVA("Aluno Aprovado") : ESCREVA("Aluno Reprovado")
```

Veja como fica em código Java:

</> Código na área

```
System.out.println( (media >= 7) ? "Aluno Aprovado" : "Aluno Reprovado" );
```

Agora, utilizando **operador ternário** para o exemplo da média com as três possibilidades de estados, temos: Aluno Aprovado, Aluno em Recuperação e Aluno Reprovado.

Esses códigos foram separados em duas linhas apenas para facilitar o entendimento.

Em código Java:

</> Código na área

```
System.out.println( (media >= 7) ? "Aluno Aprovado" :
    ((media >= 5) ? "Aluno em Recuperação" : "Aluno Reprovado") );
```

Para lembrar, **pseudocódigo** é uma forma de organizar e expressar o raciocínio lógico determinando a ordem das instruções para resolver um dado problema fazendo uso de uma linguagem natural e de fácil entendimento.

Ferramentas como Portugol e Visualg vieram para nos auxiliar, no sentido de escrever as instruções e poder ver uma execução a partir das entradas e saídas de dados. Neste caso, embora o operador condicional ternário tenha sido apresentado em pseudocódigo, nenhuma das ferramentas citadas aceita esse operador. De fato, ele é uma particularidade de algumas linguagens de programação, entre elas, Java.

A seguir, é apresentado um exemplo do uso da estrutura de controle condicional ternário. No código, a variável **situacaoEstoque** irá receber a **String** “*Produto disponível no estoque!*” se a condição (*qtd > 0*) for verdadeira. Caso contrário, a variável receberá o valor “*Produto indisponível no estoque. Efetue a reposição!*”.

</> Código na área

```
import java.util.Scanner;

public class InstrucaoCondisionalTernario {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);
        System.out.print("Nome do produto: ");
        String nome = leia.nextLine();
        System.out.print("Quantidade de produto no estoque: ");
        int qtd = leia.nextInt();
```

```

        System.out.println("**** Dados do produto ****");
        System.out.println("Produto.....: " + nome);
        System.out.println("Quantidade...: " + qtd);
        String situacaoEstoque = (qtd > 0)
            ? "Produto disponível no estoque!"
            : "Produto indisponível no estoque. Efetue a reposição!";

        System.out.println("Situação.....: " + situacaoEstoque);
    }
}

```

Se este é seu primeiro contato com linguagem de programação, pode parecer algo difícil ou que são muitas informações ao mesmo tempo. Mas não se preocupe! Você pode rever o material sempre que quiser ou entrar em contato com os tutores para tirar as suas dúvidas.

Escolha caso

A estrutura de controle **ESCOLHA .. CASO** é utilizada quando há várias condições possíveis e diferentes ações devem ser tomadas com base nessas condições. Vale ressaltar que essa estrutura é similar à **SE .. SENAO SE**, com a diferença de que seu uso mais apropriado é quando se sabe os valores a serem verificados, isto é, quando sabemos os valores que a variável da condição poderá assumir ao longo da execução.

A sintaxe da estrutura **ESCOLHA .. CASO** é apresentada a seguir:

</> Código na área

escolha // caso expressão seja igual a valor1 instruções pare	(expressão) caso valor1:
----------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------

```

        caso           valor2:
//      caso     expressão     seja     igual    a    valor2
instruções
pare

        caso           valor3:
//      caso     expressão     seja     igual    a    valor3
instruções
pare

        caso           contrario:
//      Quando     nenhum      CASO      for     satisfeito
instruções
}

```

Observe que, ao final de cada sequência de instruções, é utilizado um comando de PARE. Isso é necessário para que o programa não execute as instruções do próximo CASO. Esse comportamento pode ser útil quando queremos que uma mesma sequência de instruções seja executada em mais de um CASO.

Na linguagem de programação Java, a sintaxe dessa estrutura é:

</> Código na área

```

switch (expressão) {
    case valor1:
        instrução_1;
        break;

    case valor2:
        instrução_2;
        break;

    case valor3:
        instrução_3;
        break;

    default:
        instrução_default;
}

```

Veja um exemplo comparando esse comportamento com o comando **SE .. SENÃO**

SE:

</> Código na área

```
se (opcao == 1) {  
    escreva ("Escolheu opção 1")  
} senao (opcao == 2 ou opcao == 3) {  
    escreva("Escolheu opção 2 ou opção 3")  
} senao {  
    escreva("Escolheu outra opção")  
}
```

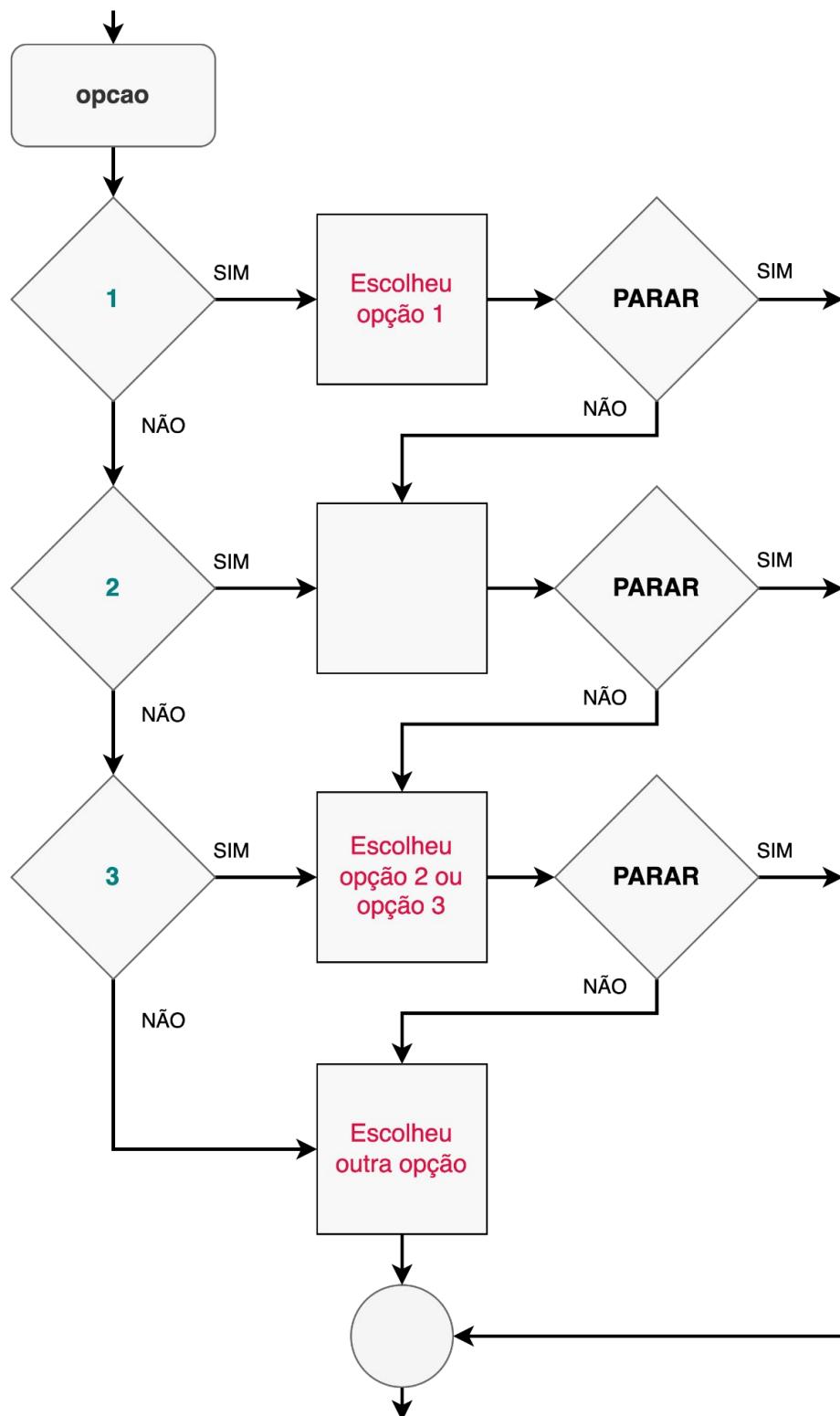
Equivalente ao comando **ESCOLHA .. CASO:**

</> Código na área

```
escolha (opcao) {  
    caso 1:  
        escreva ("Escolheu opção 1")  
        pare  
  
    caso 2:  
    caso 3:  
        escreva ("Escolheu opção 2 ou opção 3")  
        pare  
  
    caso contrario:  
        escreva("Escolheu outra opção")  
}
```

Veja o exemplo em fluxograma:

Fluxograma



Fonte: Elaborado pelos autores (2024).

Agora, em código Java:

</> Código na área

```
switch (opcao) {  
    case 1:  
        System.out.println("Escolheu opção 1");  
        break;  
    case 2:  
    case 3:  
        System.out.println("Escolheu opção 2 ou opção 3");  
        break;  
    default:  
        System.out.println("Escolheu outra opção");  
}
```

Temos, a seguir, um exemplo do uso da estrutura **ESCOLHA CASO (switch case)** em um programa Java. No exemplo, a variável `opcao` pode passar por até quatro condições e, se uma delas for satisfeita, então a respectiva instrução é executada e o comando `switch` é finalizado ao encontrar a instrução `break`. Caso nenhum `case` seja satisfeito, então a instrução `default` será executada. Em outras palavras, a instrução `default` é uma forma de finalizar o `switch` quando nenhum dos possíveis valores verificados foram satisfeitos.

Imagine um algoritmo em que o cliente de determinado comércio deverá escolher a forma de pagamento conforme as seguintes opções:



- **1** - para pagamento com **Dinheiro**
- **2** - para pagamento com **Cartão de Crédito**
- **3** - para pagamento com **Débito**
- **0** - para **Cancelar** o pagamento

Observe como fica o código:

</> Código na área

```
import java.util.Scanner;

public class InstrucaoSwitchCase {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);
        System.out.println(">>> Escolha a forma de Pagamento
<<<");
        System.out.println("1 - Dinheiro ");
        System.out.println("2 - Crédito");
        System.out.println("3 - Débito");
        System.out.println("0 - Cancelar");
        System.out.print("Opção: ");
        int opcao = leia.nextInt();

        System.out.println("*** Forma de Pagamento Escolhida
***");
        switch (opcao) {
            case 1:
                System.out.println(" * Dinheiro * ");
                break;
            case 2:
                System.out.println(" * Cartão de Crédito * ");
                break;
            case 3:
                System.out.println(" * Cartão de Débito * ");
                break;
            case 0:
                System.out.println(" * Operação Cancelada * ");
                break;
            default:
                System.out.println("A opção escolhida é
inválida!");
        }
    }
}
```

Que tal praticar um pouco?

A instrução `break` no `switch case` tem a função de finalizar a verificação dos casos. Para visualizar e entender o efeito do uso da instrução `break`, aproveite para exercitar o código apresentado, sem fazer o uso da instrução.

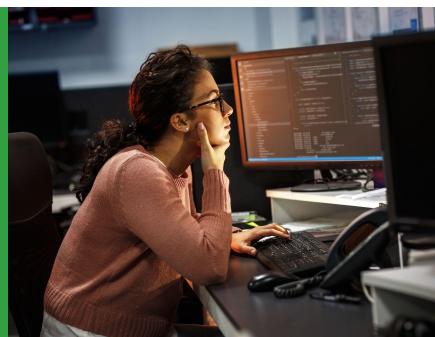
Para entender melhor as estruturas de controle, vamos fazer uma analogia com situações do nosso dia a dia.



Quando aplicar a Estrutura de Decisão "Se" e "Senão"

Analogia: decidir se você vai sair de casa ou não, dependendo do clima.

Exemplo: se estiver chovendo, você pode decidir ficar em casa (Se). Caso contrário, se estiver ensolarado, você pode decidir sair para uma caminhada (Senão).



Estrutura de Decisão "Escolha-Caso"

Analogia: escolher o que comer em um restaurante com base em um menu.

Exemplo: você olha para o menu e escolhe entre diferentes opções de pratos (Escolha). Dependendo do que você escolheu, o garçom trará o prato correspondente (Caso).



Operador Ternário (condicional ternário)

Analogia: decidir se você deve ou não levar um guarda-chuva ao sair de casa.

Exemplo: se a previsão do tempo indicar que há uma alta probabilidade de chuva, você pode decidir levar um guarda-chuva (condição ? ação se verdadeiro : ação se falso).

Conseguiu entender melhor com esses exemplos cotidianos?



Para complementar este conteúdo,
clique no botão e assista ao vídeo
disponível no canal **GV Ensino** no
YouTube.

[Acesse](#)

A partir de agora, você conhecerá as estruturas de controle de repetição. Vamos lá?

6. Estruturas de Controle de Repetição

As estruturas de controle de repetição servem para executar um bloco de código repetidamente enquanto uma determinada condição for verdadeira. Elas são fundamentais na programação, pois permitem automatizar tarefas repetitivas e iterar sobre conjuntos de dados, o que economiza tempo e reduz a quantidade necessária de código.

Em programação, essas estruturas são elaboradas a partir destas instruções:

- ENQUANTO (*WHILE*).
- FAÇA ENQUANTO (*DO WHILE*).
- PARA (*FOR*).

Cada uma dessas estruturas será explorada nas seções seguintes, e exploraremos conceito, exemplos e dicas de aplicação.

Estrutura de controle de repetição ENQUANTO

A estrutura de repetição ENQUANTO, também chamado de while nas linguagens de programação, serve para executar um bloco de código repetidamente enquanto uma condição específica permanecer verdadeira. Ela é útil em situações em que é necessário repetir uma determinada operação até que uma condição de parada seja alcançada. O controle de parada desse laço também é chamado de sentinel, pelo fato de não se saber exatamente quantas vezes as instruções do laço serão executadas.

Laço ou loop: são termos comuns para expressar um bloco de instruções que deve ser executado repetidas vezes.

01

Processamento de uma lista de dados

Pode-se usar uma repetição ENQUANTO para iterar sobre uma lista de dados e realizar operações em cada item até que um critério de parada seja atingido.

02

Interatividade com o usuário

Pode-se usar a instrução ENQUANTO para solicitar a entrada do usuário repetidamente até que uma condição específica seja atendida, por exemplo, pedir ao usuário para digitar um número entre 1 e 10.

03 Verificação de condições em tempo real

Em situações em que você precisa verificar continuamente uma condição em tempo real, como monitoramento de sensores, a estrutura ENQUANTO é útil para executar o código enquanto a condição permanecer verdadeira.

A sintaxe da estrutura **ENQUANTO .. FAÇA** é apresentada a seguir:

</> Código na área

```
enquanto (condição) {  
    //Executa enquanto a condição for verdadeira  
    intruções  
}
```

Na linguagem de programação Java, a sintaxe é a seguinte:

</> Código na área

```
while (condição) {  
    // Bloco de código a ser repetido  
}
```

O funcionamento da estrutura de repetição **while** tem como princípio duas etapas:



Verificação da condição



Execução do bloco de código

Antes de cada execução do bloco de código dentro do laço de repetição, a condição é verificada; se for verdadeira, o bloco de código é executado e, se for falsa, o loop é interrompido e a execução continua após o bloco de código. Dependendo da condição, a estrutura de repetição while poderá resultar em falso já na primeira verificação, fazendo com que as instruções do laço não sejam executadas nenhuma vez.

Se a condição for verdadeira, o bloco de código dentro do laço de repetição é executado. Após a execução do bloco de código, a condição deve ser verificada novamente. Assim, o bloco de código continuará sendo executado repetidamente até que a condição se torne falsa.

Imagine que se queira incrementar o valor de uma variável, cujo valor inicial seja 1, até que seu valor seja igual a 5. Além do incremento, também gostaríamos de mostrar na tela o novo valor.

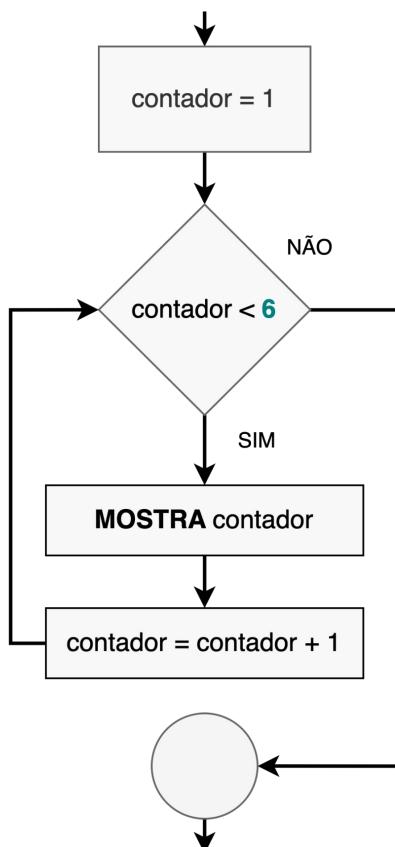
Observe este código em pseudocódigo usando o comando “enquanto faça”:

</> Código na área

```
inteiro contador = 1
enquanto (contador < 6) {
    escreva ("Contador: ", contador, "\n")
    contador = contador + 1
}
```

Veja como fica esse código usando fluxograma:

Fluxograma “enquanto faça”



Fonte: Elaborado pelos autores (2024).

Observe o código em Java:

</> Código na área

```
int contador = 1;
while (contador < 6) {
    System.out.println("Contador: " + contador);
    contador = contador + 1;
}
```

Temos, ainda, outros casos, como **laço infinito**, **interromper** e **continue**. No primeiro caso, de **LAÇO INFINITO**, podemos querer que o laço seja executado indefinidamente ou até que uma condição de parada com estado dinâmico ocorra. Assim, podemos deixar a condição de parada sempre com o valor **VERDADEIRO** e usar o comando de **INTERROMPER (break)** para sair do laço.

O comando **INTERROMPER (break)** serve para interromper um fluxo de execução. Quando utilizado dentro de um laço, o comando fará com que a execução seja interrompida e continue a execução das instruções seguintes após o laço. Esse tipo de estrutura pode ser utilizada quando temos diferentes critérios de parada ao longo do laço.

Veja como fica o código em **pseudocódigo**:

</> Código na área

```
inteiro contador = 1
enquanto (verdadeiro) {
    escreva ("Contador: ",contador,"\n")
    contador = contador + 1
    se (contador > 5) {
        pare
    }
}
```

Já o código em **Java** fica assim:

</> Código na área

```
int contador = 1;
while (true) {
    System.out.println("Contador: " + contador);
    contador = contador + 1;
    if(contador > 5) {
        break;
    }
}
```

O comando “**continue**” serve para continuar um fluxo de execução. Quando utilizado dentro de um laço, o comando fará com que a execução continue a partir do próximo laço de instruções.

Imagine que o objetivo é incrementar uma variável até 10, mas deve mostrar apenas os números pares.

Observe o código em **pseudocódigo**:

</> Código na área

```
inteiro contador = 2
enquanto (contador < 11) {
    contador = contador + 1
    resto = contador % 2
    se (resto != 0) {
        CONTINUE
    }
    escreva ("Contador: ",contador,"\n")
}
```

O código em Java fica da seguinte maneira:

</> Código na área

```
int contador = 0;
while (contador < 11) {
    contador++;
    int resto = contador % 2;
    if(resto != 0) {
        continue; //pula a iteração quando o número for
        impar
    }
    System.out.println("Contador: " + contador);
}
```

Vamos a mais um exemplo? Observe o uso da estrutura de repetição **while**. No programa, é realizada a entrada de **n** produtos (nome e quantidade). No final de cada iteração, é perguntado ao usuário se **deseja repetir** a operação (sentinela - decide se o laço de repetição será repetido ou não). Se **sim** (1), um novo produto (nome e quantidade) é registrado, a quantidade de entrada é verificada e se essa for maior que as demais então o nome do produto e a quantidade são armazenadas em variáveis auxiliares (maiorQuantidade e nomeDoProduto). Se o usuário optar por **não** repetir (2), o programa encerra o laço de repetição e segue o fluxo, exibindo o nome e a quantidade do produto de maior quantidade registrada, além do número de produtos registrados.

</> Código na área

```
import java.util.Scanner;

public class InstrucaoWhile {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);

        int maiorQuantidade = 0;
        String nomeDoProduto = null;
```

```
int opcao = 1;
int contador = 0;
while (opcao == 1) {
    contador++;
    System.out.print("Nome do produto " + contador + ": ");
    String nome = leia.next();
    System.out.print("Quantidade de produto no estoque: ");
    int qtd = leia.nextInt();

    if (qtd > maiorQuantidade) {
        maiorQuantidade = qtd;
        nomeDoProduto = nome;
    }
    System.out.println("Deseja repetir? (1) Sim (2) Não");
    System.out.print("Opção ==> ");
    opcao = leia.nextInt();
}
System.out.println("**** Dados do produto com maior estoque
****");
System.out.println("Produto.....: " +
nomeDoProduto);
System.out.println("Quantidade.....: " +
maiorQuantidade);
System.out.println("Produtos registrados..: " + contador);
}
```

A seguir, é apresentado o resultado do programa após a entrada de três produtos:

</> Código na área

```
Nome do produto 1: Tênis
Quantidade de produto no estoque: 10
Deseja repetir? (1) Sim (2) Não
Opção ==> 1
Nome do produto 2: Sapato
Quantidade de produto no estoque: 30
Deseja repetir? (1) Sim (2) Não
Opção ==> 1
Nome do produto 3: Camisa
Quantidade de produto no estoque: 20
Deseja repetir? (1) Sim (2) Não
Opção ==> 2
**** Dados do produto com maior estoque ****
Produto.....: Sapato
Quantidade.....: 30
Produtos registrados...: 3
```

Como comentado anteriormente, dependendo da condição, as instruções da repetição while podem não ser executadas nenhuma vez.

Experimente substituir o valor da variável `opcao` para **2** antes de iniciar o laço.
Execute o programa e analise o resultado.

</> Código na área

```
...
int opcao = 2;
int contador = 0;
...
```

Agora, vamos abordar a estrutura de repetição “faça enquanto”.

Estrutura de controle de repetição **FAÇA ENQUANTO**

A estrutura de repetição FAÇA ENQUANTO, também conhecida como DO WHILE na grande maioria das linguagens, é usada para repetir um bloco de código enquanto uma condição específica permanece verdadeira. Ao contrário da estrutura "enquanto", essa estrutura garante que o bloco de código seja executado pelo menos uma vez, mesmo que a condição seja inicialmente falsa. Desta forma, essa estrutura de repetição é útil quando é necessário executar um bloco de instruções pelo menos uma vez, independentemente da condição, e depois verificar se o bloco deve ser repetido com base na condição especificada.

Elá é comumente usada em situações em que a execução do bloco de código é necessária antes de verificar a condição de continuação, por exemplo, para exibir um menu de opções e após a entrada do usuário decidir se a repetição continua ou não.

A sintaxe da estrutura de controle de repetição FAÇA ENQUANTO é apresentada a seguir:

</> Código na área

```
faca {  
    // Bloco de código a ser repetido  
    instruções  
} enquanto (condição)
```

Na linguagem de programação Java, a sintaxe é a seguinte:

</> Código na área

```
do {  
    // Bloco de código a ser repetido  
    instruções;  
} while (condição);
```

O funcionamento da estrutura de repetição do while tem como princípio três etapas:

Execução inicial do bloco de código

O bloco de código dentro da estrutura "faça enquanto" é executado uma vez inicialmente, antes mesmo de verificar a condição.

1

Verificação da condição

Após a primeira execução do bloco de instruções, a condição é verificada. Se a condição for verdadeira, as instruções do laço serão executadas novamente. Caso contrário, a repetição será encerrada e as instruções após o bloco "faça enquanto" (do while) serão executadas.

Continuação da execução

Se a condição for verdadeira, o bloco de instruções pertencentes ao do while será executado novamente e o processo de verificação da condição continuará. Isso ocorre até que a condição se torne falsa.

2

3

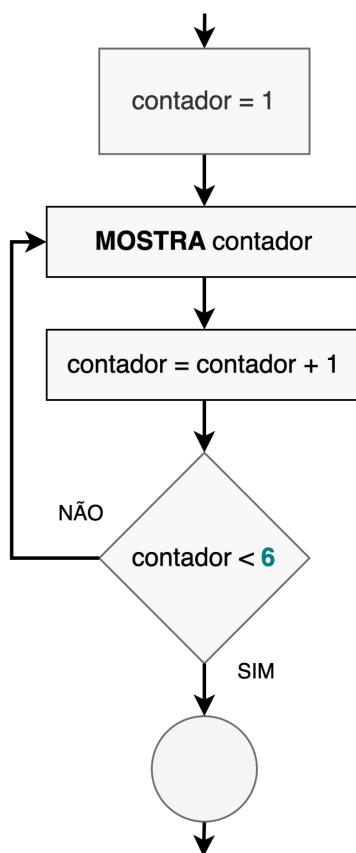
Observe o código do contador apresentado anteriormente:

</> Código na área

```
inteiro contador = 1
faca {
    escreva (contador)
    contador++;
} enquanto (contador < 6)
```

Agora, como fica a solução usando **fluxograma**:

Fluxograma *while*



Fonte: Elaborado pelos autores (2024).

O código em **Java** é:

</> Código na área

```
int contador = 1;
do {
    System.out.println("Contador: " + contador);
    contador = contador + 1;
} while (contador < 6);
```

A seguir, confira um exemplo do uso da estrutura de repetição **while**. No programa, é realizada a entrada de **n** produtos (nome e quantidade). No final de cada iteração, é perguntado ao usuário se **deseja repetir** a operação. Se **sim** (1), um novo produto (nome e quantidade) é registrado, a quantidade de entrada é verificada e, se ela for maior que as demais, então, o nome do produto e a quantidade são armazenadas em variáveis auxiliares (maiorQuantidade e nomeDoProduto). Se o usuário optar por **não repetir** (2), o programa encerra o laço de repetição e segue o fluxo, exibindo o nome e a quantidade do produto de maior quantidade registrada, além do número de produtos registrados.

</> Código na área

```
import java.util.Scanner;

public class InstrucaoDoWhile {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);

        int maiorQuantidade = 0;
        String nomeDoProduto = null;

        int opcao = 0;
        int contador = 0;
        do {
            contador++;
            System.out.print("Nome do produto " + contador + ": ");
            String nome = leia.next();
            System.out.print("Quantidade de produto no estoque: ");
            int qtd = leia.nextInt();
        } while (opcao == 1);
    }
}
```



```
        if (qtd > maiorQuantidade) {
            maiorQuantidade = qtd;
            nomeDoProduto = nome;
        }
        System.out.println("Deseja repetir? (1) Sim (2) Não");
        System.out.print("Opção ==> ");
        opcao = leia.nextInt();
    } while (opcao == 1);

    System.out.println("**** Dados do produto com maior estoque ****");
    System.out.println("Produto.....: " + nomeDoProduto);
    System.out.println("Quantidade.....: " + maiorQuantidade);
    System.out.println("Produtos registrados...: " + contador);
}
}
```

O resultado do programa após a entrada de três produtos será:

</> Código na área

```
Nome do produto 1: Tênis
Quantidade de produto no estoque: 10
Deseja repetir? (1) Sim (2) Não
Opção ==> 1
Nome do produto 2: Sapato
Quantidade de produto no estoque: 40
Deseja repetir? (1) Sim (2) Não
Opção ==> 1
Nome do produto 3: Camisa
Quantidade de produto no estoque: 25
Deseja repetir? (1) Sim (2) Não
Opção ==> 2
**** Dados do produto com maior estoque ****
Produto.....: Sapato
Quantidade.....: 40
Produtos registrados...: 3
```

Como já mencionado, a estrutura de repetição do *while* tem como principal característica executar as instruções do laço pelo menos uma vez. Experimente substituir o valor da variável *opcao* para 1 antes de iniciar o laço. Execute o programa e analise o resultado e compare com o código implementado na estrutura *while* da seção “Código na área” do tema que trata do laço “enquanto faça”.

</> Código na área

```
...
int opcao = 1;
int contador = 0;
...
```

Até agora utilizamos instruções condicionais em que foram avaliados números. Mas como se comportaria o programa se a variável utilizada na condição fosse uma String e você desejasse comparar a igualdade dela com outra? Primeiramente, experimente o código a seguir, implemente, execute as entradas e analise os resultados. O programa pede para digitar uma senha e, pela lógica, deve executar até que a senha correta (“**senha123**”) seja digitada:

</> Código na área

```
import java.util.Scanner;

public class InstrucaoDoWhileComStringErrada {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String senhaDigitada;

        do {
            System.out.print("Digite sua senha: ");
            senhaDigitada = scanner.nextLine();
        } while (senhaDigitada != "senha123");

        System.out.println("Senha correta! Acesso permitido.");
    }
}
```

Você deve ter percebido que a execução não tem fim, mesmo quando a senha correta é digitada.

Agora, faça o código a seguir e experimente novamente.

</> Código na área

```
import java.util.Scanner;

public class InstrucaoDoWhileComString {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String senhaDigitada;

        do {
            System.out.print("Digite sua senha: ");
            senhaDigitada = scanner.nextLine();
        } while (!senhaDigitada.equals("senha123"));

        System.out.println("Senha correta! Acesso permitido.");
    }
}
```

Bom, feito isso, qual a explicação para o ocorrido?

O fato é que uma String em Java é uma classe (conceito da orientação a objetos, que será tratado no próximo semestre) e uma variável desse tipo se trata de um objeto. Logo, a comparação do seu conteúdo com os operadores relacionais (`==`, `<=`, `>=` e `!=`) não tem o mesmo efeito que um tipo primitivo (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`), por isso, para comparar o conteúdo de uma String, é necessário fazer o uso do método `equals` ou `equalsIgnoreCase`.

Ambos os métodos têm por características comparar duas Strings, sendo elas o conteúdo de uma String originária (`senhaDigitada`) e uma segunda String (`"senha123"`). Caso as Strings sejam iguais, o retorno será verdadeiro (`true`), o que

satisfaz à condição e faz com que o bloco de comandos da instrução **while** seja executado, do contrário, a instrução é encerrada. Considerando que **equals** e **equalsIgnoreCase** resultam em um valor booleano (**true** ou **false**) após a verificação, esses métodos podem ser utilizados em instruções condicionais também (**if else**).

A diferença entre os dois métodos é que o segundo desconsidera (ignora) a diferenciação das letras minúsculas e maiúsculas, isto é, naturalmente “**senha123**” é diferente de “**SENHA123**”, mas, usando **equalsIgnoreCase**, esse detalhe não será considerado.

Agora, é hora de abordarmos a estrutura de repetição PARA. Vamos lá?

Estrutura de controle de repetição **PARA**

A estrutura de controle de repetição PARA, denominada for, na maioria das linguagens de programação, é uma das estruturas de controle de fluxo mais utilizadas em programação.

Essa instrução é muito útil quando se sabe antecipadamente quantas vezes um bloco de código precisa ser repetido.

A estrutura de repetição PARA é usada para executar um número específico de vezes um determinado bloco de instruções. Ela permite iterar sobre uma sequência de valores, como números inteiros, avançando ou retrocedendo em uma determinada

sequência. Isso é útil quando se deseja repetir uma ação um número conhecido de vezes.

A sintaxe da estrutura **PARA** é apresentada a seguir:

</> Código na área

```
para (inicialização; condição; incremento_decremento) {  
    // Bloco de código a ser repetido  
    intruções  
}
```

Na linguagem de programação Java, a sintaxe dessa instrução é:

</> Código na área

```
for (inicialização; condição; incremento_decremento) {  
    // Bloco de código a ser repetido  
    intruções;  
}
```

Na estrutura de repetição for, o que significa inicialização, condição e incremento? Acompanhe!

01 Inicialização

Espaço em que são declaradas as variáveis de controle, podendo ser declaradas e inicializadas uma ou mais variáveis. O código inserido nessa área será executado uma única vez antes de se iniciar a execução do bloco de instruções do laço. Geralmente, essa variável é usada para controlar a quantidade de repetições do laço (loop).

02 Condição

É uma estrutura que controla a continuação do loop, isto é, a cada iteração, a condição é verificada. Enquanto a condição for verdadeira, o bloco de instruções dentro do loop é executado.

03 Incremento ou decremento

Também chamada de expressão de controle, nesse espaço, é colocado o tamanho do passo da iteração (1 em 1, 2 em 2, 3 em 3, ou outra forma de determinar o avanço da variável de controle).

Após cada execução do bloco de instruções, a variável de controle é atualizada (incrementada ou decrementada) para garantir que a condição de continuação possa eventualmente se tornar falsa para encerrar o loop. Vale destacar que se o valor da variável de controle não encontrar a **condição falsa**, o laço executará infinitamente.

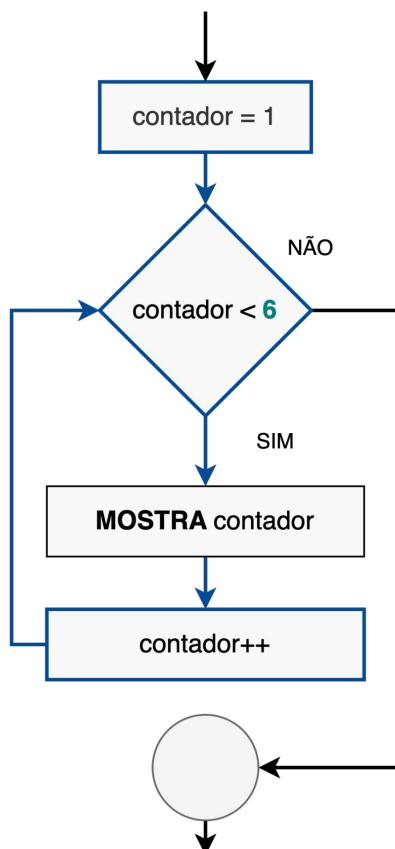
Veja como fica o código do contador apresentado anteriormente:

</> Código na área

```
inteiro contador
para (contador=1; contador<6; contador++) {
    // Bloco de código a ser repetido
    escreva ("Contador: ", contador, "\n")
}
```

Já a representação do comando “para” no **fluxograma** fica da seguinte forma:

Fluxograma PARA



Fonte: Elaborado pelos autores (2024).

A implementação em **Java** fica desta maneira:

</> Código na área

```
for(int contador = 1; contador < 6; contador++){  
    System.out.println("Contador: " + contador);  
}
```

Observe, agora, um exemplo do uso da estrutura de repetição for. No programa, é realizada a entrada de cinco produtos (nome e quantidade). Em cada iteração, é verificada a quantidade de entrada, e a maior delas é armazenada. No final da

execução, o programa exibe o nome e a quantidade do produto de maior quantidade registrada.

</> Código na área

```
import java.util.Scanner;

public class InstrucaoFor {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);

        int maiorQuantidade = 0;
        String nomeDoProduto = null;

        for (int i = 0; i < 5; i++) {
            System.out.print("Nome do produto: ");
            String nome = leia.next();
            System.out.print("Quantidade de produto no estoque:");
        );
        int qtd = leia.nextInt();

        if (qtd > maiorQuantidade) {
            maiorQuantidade = qtd;
            nomeDoProduto = nome;
        }
    }
    System.out.println("**** Dados do produto com maior
estoque ****");
    System.out.println("Produto.....: " + nomeDoProduto);
    System.out.println("Quantidade...: " +
maiorQuantidade);
}
}
```

O resultado do programa após a entrada de cinco produtos será:

</> Código na área

```
Nome do produto: Tenis
Quantidade de produto no estoque: 10
Nome do produto: Sapato
Quantidade de produto no estoque: 30
Nome do produto: Bermuda
Quantidade de produto no estoque: 20
Nome do produto: Camisa
Quantidade de produto no estoque: 50
Nome do produto: Camiseta
Quantidade de produto no estoque: 5
**** Dados do produto com maior estoque ****
Produto.....: Camisa
Quantidade...: 50
```

Finalizado o conteúdo sobre estruturas de repetição, o próximo assunto a ser abordado são as estruturas de dados homogêneas. Confira adiante.

7. Variáveis compostas homogêneas

As estruturas de dados homogêneas são tecnicamente conhecidas como **Arrays**.

Elas se dividem em **vetores** e **matrizes**. Essas estruturas servem para armazenar um número fixo de valores que sejam de um mesmo tipo de dados, por isso, são homogêneas.

Os vetores são estruturas unidimensionais, enquanto as matrizes são multidimensionais, sendo as mais comuns as matrizes bidimensionais. Cada um desses tipos será explorado nas subseções a seguir.

Vetores - Arrays unidimensionais

Vetores, ou arrays unidimensionais, são estruturas de dados que armazenam uma coleção de elementos do mesmo tipo de dados. Eles são úteis para armazenar múltiplos valores relacionados sob um único nome.

Imagine, por exemplo, armazenar cinco quantidades de produtos. Em um primeiro momento, o que nos vem à mente é criar cinco variáveis para isso.

Vetores: são utilizados para armazenar uma lista organizada de elementos do mesmo tipo de dados, facilitando o acesso e a manipulação desses elementos.



</> Código na área

```
int qtd1, qtd2, qtd3, qtd4, qtd5;
```

Imagine, agora, se fossem 50, 500 ou mais. Você faria uma variável para cada situação? Isso seria uma solução praticamente inviável e, ao mesmo tempo, impraticável, ou seja, **não seria** uma boa prática de programação. Deste modo, a solução é criar uma única variável capaz de armazenar **vários dados** de um **mesmo tipo**, a partir da qual, por meio de um único nome, podemos acessar cada valor (elemento).

Em pseudocódigo, a declaração de uma variável vetor é ocorre seguinte forma:

</> Código na área

```
vetor NomeDoVetor[Tamanho] ;
```

Na linguagem de programação Java, uma variável do tipo vetor deve ser declarada da seguinte maneira:

</> Código na área

```
TipoDoElemento[] nomeDoVetor = new TipoDoElemento[Tamanho];
```

O que significa cada elemento da declaração?

Primeiramente, é importante saber que o que está à esquerda do sinal “=” corresponde à declaração do vetor e do tipo de dado que poderá ser armazenado nele. O que está à direita se trata da criação de um objeto vetor na memória. Para entender melhor, a seguir será explicado cada elemento que compõe a declaração e a criação do vetor.

TipoDoElemento []

É o tipo de dado dos elementos do vetor. Pode ser qualquer tipo de dados válido em Java, como **int**, **double**, **String**, ou até mesmo um tipo de objeto. Os colchetes após o **TipoDoElemento** caracterizam exatamente a declaração de um vetor.

nomeDoVetor

É o nome dado ao vetor, por meio do qual se faz o acesso e a manipulação de cada elemento (conteúdo) do vetor ao longo do programa.

new

É utilizado para criar o vetor na memória.

TipoDoElemento

É utilizado após a palavra reservada **new**.

Trata-se do tipo de dado que será utilizado para criar o objeto vetor.

Tamanho

É o número de elementos (capacidade) do vetor e deve ser envolvido entre colchetes, por exemplo, **[50]**.

=

O sinal de igual é utilizado para atribuição de valor, e indica que a instrução à sua direita está criando um objeto vetor na memória e que o endereço de memória em que o vetor foi alocado será atribuído à variável **nomeDoVetor** que está à esquerda do sinal **=**, a qual será a referência (variável de referência) para acessar o conteúdo do vetor.

A declaração e criação do vetor também pode ser realizada em duas etapas, em momentos distintos, mas mantendo a ordem declaração seguida da criação. Veja a sintaxe a seguir:

</> Código na área

```
TypeDef[] nomeDoVetor;  
// outros códigos antes de utilizar o vetor  
nomeDoVetor = new TipoDoElemento[Tamanho];
```

Uma **observação** bem importante é que, para utilizar o vetor, ele precisa obrigatoriamente ter sido criado.

Compreendida a sintaxe de um vetor, apresentamos a seguir alguns exemplos de declarações de vetor em Java:

</> Código na área

```
//declaração e criação de um vetor de 10 números inteiros  
int[] vetor = new int[10];  
  
//declaração e criação de um vetor de 20 preços de produtos  
float[] precos = new float[20];  
  
//declaração de um vetor de nomes de alunos  
String[] alunos;  
  
//criação do vetor com capacidade para armazenar 40 nomes  
alunos = new String[40];
```

Agora que vimos como declarar e criar vetores, vamos analisar como é realizado o acesso aos elementos do vetor. O vetor tem um único nome, porém, pode armazenar vários valores. O acesso a cada valor se dá por meio de um **índice** de referência para cada elemento. Esse **índice** é dado por um número que começa em **0 (primeiro elemento)** e vai até o **tamanho do vetor menos um (último elemento)**. Isto é, se o vetor foi declarado com tamanho de **10** elementos, então, o acesso de cada elemento se dá pelos **índices** de **0 a 9**. A seguir, é possível observar algumas situações de acesso ao conteúdo de um vetor:

</> Código na área

```
int[] notas = new int[5]; //declaração e criação do vetor notas
```

Neste momento, todas as **notas** serão inicializadas com o valor **0** a criar o vetor. Na memória, a situação do vetor será a seguinte:

Notas na memória

notas =	0	0	0	0	0
[índice] ==>	0	1	2	3	4

Fonte: Elaborada pelos autores (2024).

Para inserir um valor em uma determinada posição do vetor, é necessário seguir esta sintaxe:

</> Código na área

```
nome_vetor[índice] = valor;
```

O **índice** entre colchetes se refere à posição do vetor. Para fins de demonstração, a seguir, serão realizadas algumas inserções no vetor.

</> Código na área

```
.
.
.
notas[1] = 10;//atribuindo o valor 10 na segunda posição do vetor
notas[4] = 8;//atribuindo o valor 8 na quinta posição do vetor
int nota = 9;
int ind = 2;
//na instrução abaixo o valor da variável nota (9)
//será colocado na posição ind (2) do vetor
notas[ind] = nota;
.
.
```

Após as inserções realizadas nos códigos apresentados, o conteúdo do vetor **notas** na memória será conforme mostra esta figura :

Vetor “notas” na memória

notas =	0	10	9	0	8
[índice] ==>	0	1	2	3	4

Fonte: Elaborada pelos autores (2024).

Seguindo essa linha de raciocínio, a seguir, são mostradas as instruções para obter algum valor específico do vetor (considere ainda as linhas de códigos anteriores).

</> Código na área

```
System.out.println("Nota do segundo aluno: " + notas[1]);
System.out.println("Nota do quinto aluno: " + notas[4]);
System.out.println("Nota do aluno " + (ind+1) + ": " + 
notas[ind]);
int mediaDasNotas =
(notas[0] + notas[1] + notas[2] + notas[3] + notas[4]) / 5;
System.out.println("Média das notas: " + mediaDasNotas);
```

Após escrever a instrução para calcular `mediaDasNotas`, você pode estar se perguntando, “e se fossem **40 notas**, teria de escrever a soma de cada uma delas?”.

</> Código na área

```
int mediaDasNotas =
(notas[0] + notas[1] + . . . + notas[38] + notas[39]) / 40;
```

Fazer uma instrução como essa é realmente impraticável, isto é, não estaríamos aplicando de forma apropriada os recursos da linguagem de programação e tampouco os conhecimentos de lógica de programação. Enfim, nota-se aí que há um processo repetitivo, que é a soma de **n** notas. Nesse caso, o mais apropriado é fazer uso de uma instrução de repetição para fazer a iteração entre as notas. No exemplo a seguir, vamos resolver a média desta forma:

</> Código na área

```
int somaDasNotas = 0;  
  
for (int i = 0; i < 5; i++) {  
    somaDasNotas += notas[i];  
}  
  
int mediaDasNotas = somaDasNotas / 5;  
System.out.println("Média das notas: " + mediaDasNotas);
```

No código anterior, a variável **i** (índice) serve de controle para acessar cada elemento do vetor em cada iteração do laço de repetição. A condição **i < 5** serve para verificar se o final do vetor já foi atingido e, caso seja falsa a repetição para e, teremos a soma de todas as notas. Usar o literal **5**, tanto na condição quanto na instrução, para calcular a média, acaba sendo uma prática indesejável, pois, se mudássemos o tamanho do vetor, teríamos que tomar o cuidado de alterar todo o código nas situações em que o tamanho do vetor é utilizado como referência. Assim, podemos melhorar o código fazendo uma declaração de uma constante (**TAM**) para definir o tamanho do vetor e usar como referência no programa. A seguir, veja um exemplo:

</> Código na área

```
final int TAM = 5;// uma constante chamada TAM
int[] notas = new int[TAM];//TAM será usada para definir o tamanho
int somaDasNotas = 0;
for (int i = 0; i < TAM; i++) { //TAM sendo usado na condição
    somaDasNotas += notas[i];
}

//abaixo TAM sendo usada para calcular
int mediaDasNotas = somaDasNotas / TAM;
System.out.println("Média das notas: " + mediaDasNotas);
```

Outra maneira ainda, para conhecer o **tamanho** e os limites de um vetor, é utilizar a propriedade **length** da classe **Arrays**. A propriedade **length** mantém um valor que corresponde ao **tamanho** do **vetor**, isto é, a quantidade de elementos que podem ser armazenados no vetor. No código seguinte, é possível observar o uso de **length**.

</> Código na área

```
final int TAM = 5;// uma constante chamada TAM
int[] notas = new int[TAM];//TAM será usada para definir o tamanho
int somaDasNotas = 0;
for (int i = 0; i < notas.length; i++) {
    somaDasNotas += notas[i];
}

int mediaDasNotas = somaDasNotas / notas.length;//uso de length
System.out.println("Média das notas: " + mediaDasNotas);
```

Anote esta **dica**: na linguagem de programação Java, há uma forma de aplicar a estrutura de repetição **for** para fazer a iteração entre os elementos do vetor que é chamado de **for aprimorado (foreach)** em algumas linguagens de programação). Essa estrutura é mais simples de usar para os casos em que se deseja percorrer os elementos do início até o fim do vetor (ou até uma condição de parada). A sintaxe dessa estrutura é a seguinte:

</> Código na área

```
for (tipo_de_dado variavel : nome_do_vetor) {  
    //bloco de instruções  
}
```

O que significa cada instrução do **for aprimorado**?

tipo_de_dado
É o tipo de dado do vetor.



variavel
É uma variável que vai receber o elemento de cada iteração do laço de repetição, isto é, a **variável** vai receber o primeiro elemento na primeira iteração; o segundo elemento, na segunda iteração, e assim sucessivamente, até finalizar a repetição quando encontrar o último elemento do vetor.

nome_do_vetor
Trata-se do vetor que cujos dados desejamos manipular.



A seguir, será mostrado o laço de repetição **for aprimorado** para calcular a média das notas do vetor **notas**:

</> Código na área

```
final int TAM = 5;
int[] notas = new int[TAM];
int somaDasNotas = 0;
for (int nota : notas) { //aplicação do for aprimorado
    somaDasNotas += nota;
}
int mediaDasNotas = somaDasNotas / notas.length;//uso de length
System.out.println("Média das notas: " + mediaDasNotas);
```

O programa a seguir trata de um vetor para armazenar a quantidade vendida de **n** produtos. Inicialmente, é solicitada a inserção das quantidades vendidas de cada produto e, então, o conteúdo do **vetor vendas** é apresentado ao usuário:

</> Código na área

```
import java.util.Scanner;

public class VetorVendas {
    public static void main(String[] args) {

        final int TAM = 5;
        int[] vendas = new int[TAM];
        System.out.println("Tamanho do vetor de vendas: " + vendas.length);
        Scanner leia = new Scanner(System.in);
        System.out.println(">>> Registro de vendas de produtos <<<");
        for (int i = 0; i < vendas.length; i++) {
            System.out.print("Produto " + (i+1) + ": ");
            vendas[i] = leia.nextInt();
        }
    }
}
```

```
//mostrando o conteúdo do vetor de vendas
System.out.println("\n**** Quantidades Vendidas ****");
for (int i = 0; i < vendas.length; i++) {
    System.out.println("Produto " + (i+1) + ": " + vendas[i]);
}
}
```

O resultado do programa após a entrada da quantidade de cinco produtos é o seguinte:

</> Código na área

```
Tamanho do vetor de vendas: 5
>>> Registro de vendas de produtos <<<
Produto 1: 15
Produto 2: 30
Produto 3: 28
Produto 4: 10
Produto 5: 65

**** Quantidades Vendidas ****
Produto 1: 15
Produto 2: 30
Produto 3: 28
Produto 4: 10
Produto 5: 65
```

O código a seguir apresenta a manipulação dos dados do vetor. Além de uma rotina para mostrar o conteúdo, há uma rotina para calcular a média de vendas de produtos e outra para apresentar a maior quantidade vendida.

</> Código na área

```
import java.util.Scanner;

public class VetorVendasTratandoDados {
    public static void main(String[] args) {

        final int TAM = 5;
        int[] vendas = new int[TAM];
        System.out.println("Tamanho do vetor de vendas: " + vendas.length);

        Scanner leia = new Scanner(System.in);
        System.out.println(">>> Registro de vendas de produtos <<<");
        for (int i = 0; i < vendas.length; i++) {
            System.out.print("Produto " + (i+1) + ": ");
            vendas[i] = leia.nextInt();
        }

        //mostrando o conteúdo do vetor de vendas
        System.out.println("\n**** Quantidades Vendidas ****");
        for (int i = 0; i < vendas.length; i++) {
            System.out.println("Produto " + (i+1) + ": " + vendas[i]);
        }

        //Média de vendas
        System.out.println("\n**** Média de Vendas ****");
        int soma = 0;
        for (int i = 0; i < vendas.length; i++) {
            soma += vendas[i];
        }
        System.out.println("Média de vendas: " + (soma / vendas.length));

        //Encontrando o produto mais vendido
        System.out.println("\n**** Maior quantidade vendida ****");
        int maior = vendas[0];
        for (int i = 1; i < vendas.length; i++) {
            if (vendas[i] > maior) {
                maior = vendas[i];
            }
        }
        System.out.println("A maior quantidade vendida foi: " + maior);
    }
}
```

A partir da entrada das quantidades de produto, o programa anterior apresenta as seguintes saídas:

</> Código na área

```
Tamanho do vetor de vendas: 5
>>> Registro de vendas de produtos <<<
Produto 1: 15
Produto 2: 85
Produto 3: 35
Produto 4: 20
Produto 5: 45

**** Quantidades Vendidas ****
Produto 1: 15
Produto 2: 85
Produto 3: 35
Produto 4: 20
Produto 5: 45

**** Média de Vendas ****
Média de vendas: 40

**** Maior quantidade vendida ****
A maior quantidade vendida foi: 85
```

Para finalizar, veja a implementação do mesmo código, porém, aplicando-se o **for aprimorado**:

</> Código na área

```
import java.util.Scanner;

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

```
final int TAM = 5;
int[] vendas = new int[TAM];
System.out.println("Tamanho do vetor de vendas: " + vendas.length);

Scanner leia = new Scanner(System.in);
System.out.println(">>> Registro de vendas de produtos <<<");
for (int i = 0; i < vendas.length; i++) {
    System.out.print("Produto " + (i+1) + ": ");
    vendas[i] = leia.nextInt();
}

//mostrando o conteúdo do vetor de vendas
System.out.println("\n**** Quantidades Vendidas ****");
for (int i = 0; i < vendas.length; i++) {
    System.out.println("Produto " + (i+1) + ": " + vendas[i]);
}

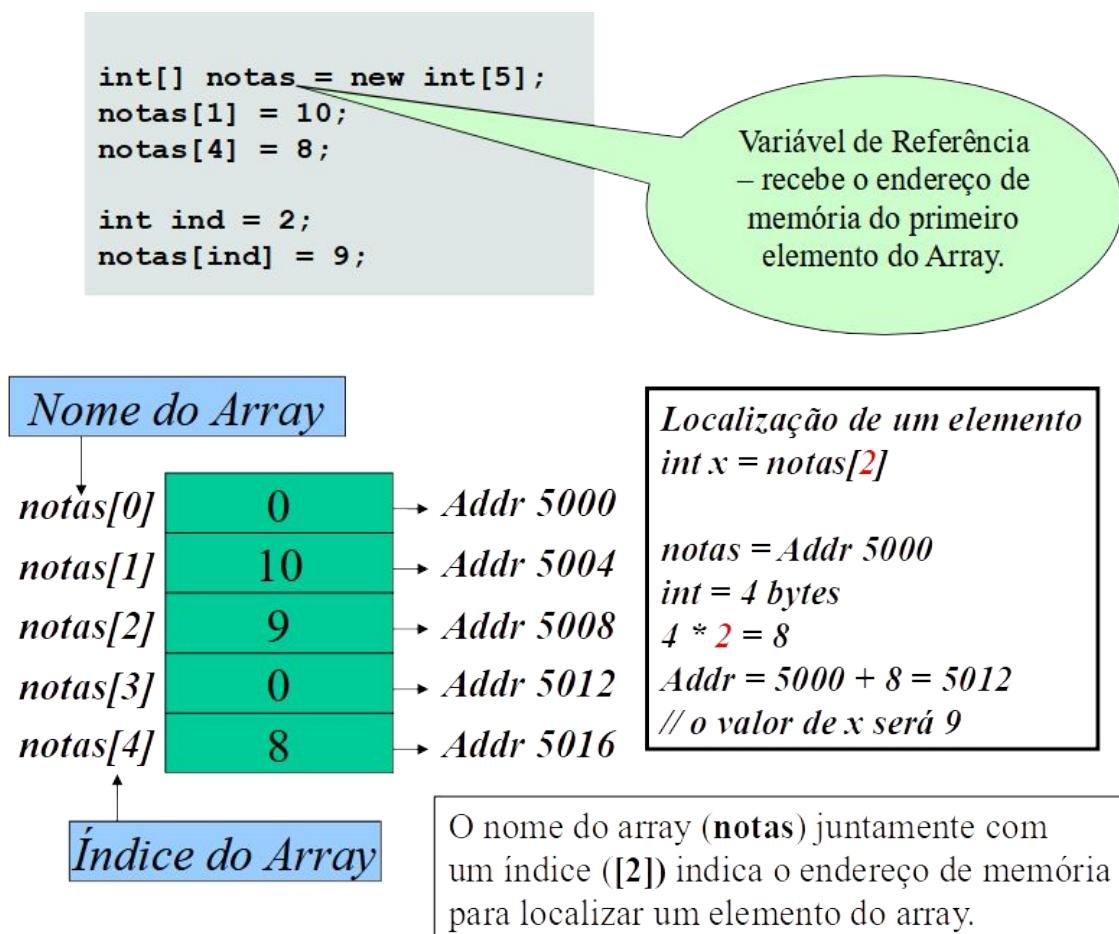
//Média de vendas
System.out.println("\n**** Média de Vendas ****");
int soma = 0;
for (int quantidade : vendas) {
    soma += quantidade;
}
System.out.println("Média de vendas: " + (soma / vendas.length));

//Encontrando o produto mais vendido
System.out.println("\n**** Maior quantidade vendida ****");
int maior = vendas[0];
for (int quantidade : vendas) {
    if (quantidade > maior) {
        maior = quantidade;
    }
}
System.out.println("A maior quantidade vendida foi: " + maior);
}
```

Para uma compreensão melhor, a figura a seguir mostra uma representação do **vetor notas** na memória e como seus dados são manipulados a partir do nome do vetor. Vale destacar que, na figura, o **valor 5000** para **Addr** (Address - endereço de memória) é simbólico, haja vista que os endereços de memória são alocados em locais disponíveis em tempo de execução e, por isso, são dinâmicos, variando a cada vez que o programa é executado.



Representação do vetor notas



Fonte: Elaborada pelos autores (2024).

Neste tópico, abordamos as estruturas de arrays unidimensionais. A partir de agora, vamos tratar sobre as estruturas de arrays multidimensionais.

Matrizes - Arrays multidimensionais

As matrizes, ou arrays multidimensionais, são estruturas de dados que armazenam coleções de elementos em múltiplas dimensões. Elas são úteis para representar dados tabulares, como tabelas e planilhas, isto é, quando é necessário representar dados em duas ou mais dimensões.

Arrays multidimensionais, e isso também vale para a linguagem de programação Java, permitem armazenar dados em mais de uma dimensão, o que é útil para representar informações tabulares, matrizes, ou até mesmo volumes de dados tridimensionais.

Os dois tipos mais comuns de arrays multidimensionais em Java são os arrays bidimensionais e tridimensionais, mas, neste estudo, vamos nos concentrar nos arrays bidimensionais.

Um array bidimensional (matriz bidimensional) é uma estrutura de dados que organiza elementos em linhas e colunas, formando uma matriz retangular. Cada elemento em um array bidimensional é acessado por um par de índices, representando a linha e a coluna em que o elemento está localizado.

A sintaxe para declarar e criar um array bidimensional em Java é a seguinte:

</> Código na área

```
TipoDoElemento[][] nomeDoArray = new TipoDoElemento[linhas][colunas];
```

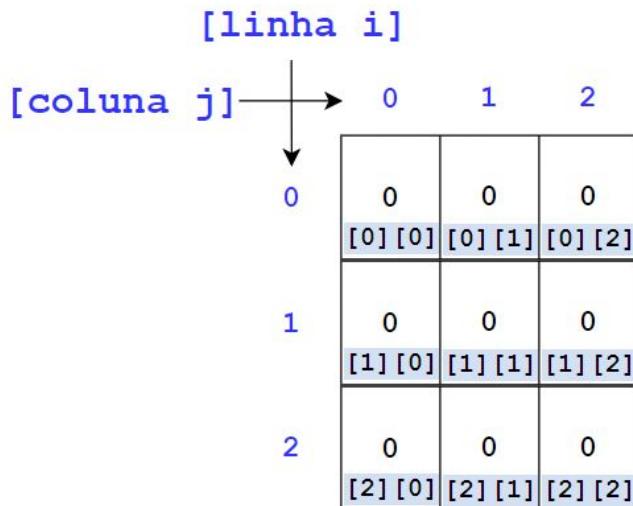
O que difere a declaração de um array bidimensional de um unidimensional é o conjunto é a dupla de colchetes [][], um para definir as linhas, e outro, as colunas.

A figura a seguir apresenta a declaração de uma matriz m e mostra como seria sua

representação na memória após a criação. No exemplo, é criada uma matriz quadrada 3 x 3, isto é, 3 linhas e 3 colunas.

Declaração de uma matriz m

```
int[][] m = new int[3][3];
```

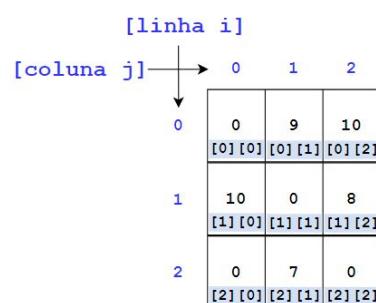


Fonte: Elaborada pelos autores (2024).

Cada elemento da matriz é identificado pelos índices **i** (linha) e **j** (coluna). Para acessar um determinado elemento da matriz **m**, temos de informar a referência **[i]** e **[j]**. A figura seguinte mostra o conteúdo da matriz após a inserção de valores em algumas posições da matriz:

Conteúdo da matriz

```
int[][] m = new int[3][3];
m[0][1] = 9;
m[1][0] = 10;
m[1][2] = 8;
m[2][1] = 7;
int i = 0;
int j = 2;
int valor = 10;
m[i][j] = valor;
```



Fonte: Elaborada pelos autores (2024).

A seguir, para exemplificar, vamos declarar um array bidimensional para armazenar as **notas de 2 bimestres de 5 alunos**. Para isso, vamos assumir que as linhas representam cada aluno (*índice i de 0 a 4*), enquanto as colunas serão utilizadas para representar cada bimestre (*índice j de 0 a 1*).

</> Código na área

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

A navegação (iteração) pelo conteúdo da matriz requer, geralmente, a implementação de um laço **for** dentro de outro, ou seja, o uso de **for aninhado**. Em sequência, é apresentado o código para fazer a entrada das notas dos alunos para cada bimestre:

</> Código na área

```
System.out.println(">>>> Entrada das notas dos alunos <<<<");  
for (int i = 0; i < 5; i++) {  
    System.out.println("Notas do aluno " + (i + 1));  
    for (int j = 0; j < 2; j++) {  
        System.out.print("Nota do " + (j+1) + "º Bimestre: ");  
        notas[i][j] = leia.nextInt();  
    }  
}
```

Para acessar um elemento específico da matriz, é necessário indicar a linha(i) e a coluna(j):

</> Código na área

```
notas[i][j] = leia.nextInt();
```

Por exemplo, para exibir o conteúdo da matriz de notas, o seguinte código pode ser implementado:

</> Código na área

```
//mostrando as notas dos alunos
System.out.println("\n***** Notas dos Alunos *****");
for (int i = 0; i < 5; i++) {
    System.out.println("Notas do aluno " + (i + 1));
    for (int j = 0; j < 2; j++) {
        System.out.println("Nota do " + (j+1)
            + "º Bimestre: " + notas[i][j]);
    }
    System.out.println();
}
```

O programa a seguir trata de uma matriz para armazenar a quantidade vendida de **cinco** produtos nos últimos dois anos. O programa apresenta três rotinas bem definidas:

01

Inicialmente, é solicitada a inserção das quantidades vendidas de cada produto para cada ano.

02

O conteúdo da matriz de vendas dos produtos é apresentado.

03

O total de vendas de cada produto é calculado e mostrado.

Agora, veja o código:

</> Código na área

```
import java.util.Scanner;

public class MatrizVendas {
    public static void main(String[] args) {
        Scanner leia = new Scanner(System.in);

        int[][] vendas = new int[5][2];

        System.out.println("">>>> Entrada das vendas dos produtos <<<<");
        System.out.println("">>>>> Dos últimos dois anos <<<<");
        for (int i = 0; i < 5; i++) {
            System.out.println("Vendas do produto " + (i + 1));
            for (int j = 0; j < 2; j++) {
                System.out.print("Ano " + (j+1) + ": ");
                vendas[i][j] = leia.nextInt();
            }
        }

        //mostrando as vendas dos produtos
        System.out.println("\n**** Vendas dos Produtos ****");
        for (int i = 0; i < 5; i++) {
            System.out.println("Vendas do produto " + (i + 1));
            for (int j = 0; j < 2; j++) {
                System.out.println("Ano " + (j+1) + ": " + vendas[i][j]);
            }
            System.out.println();
        }

        //mostrando o total de vendas por produto
        System.out.println("\n**** Total de vendas por produtos ****");
        for (int i = 0; i < 5; i++) {
            int soma = 0;
            for (int j = 0; j < 2; j++) {
                soma += vendas[i][j];
            }
            System.out.println("Vendas do produto " + (i + 1)
                + ": " + soma + "\n");
        }

    }
}
```

Após a entrada de dados, o programa exibirá o seguinte resultado:

</> Código na área

```
>>> Entrada das vendas dos produtos <<<  
>>> Dos últimos dois anos <<<  
Vendas do produto 1  
Ano 1: 100  
Ano 2: 150  
Vendas do produto 2  
Ano 1: 200  
Ano 2: 220  
Vendas do produto 3  
Ano 1: 50  
Ano 2: 80  
Vendas do produto 4  
Ano 1: 300  
Ano 2: 280  
Vendas do produto 5  
Ano 1: 125  
Ano 2: 200  
  
**** Vendas dos Produtos ****  
Vendas do produto 1  
Ano 1: 100  
Ano 2: 150  
  
Vendas do produto 2  
Ano 1: 200  
Ano 2: 220  
  
Vendas do produto 3  
Ano 1: 50  
Ano 2: 80  
  
Vendas do produto 4  
Ano 1: 300  
Ano 2: 280  
  
Vendas do produto 5  
Ano 1: 125  
Ano 2: 200  
  
**** Total de vendas por produtos ****  
Vendas do produto 1: 250  
Vendas do produto 2: 420  
Vendas do produto 3: 130  
Vendas do produto 4: 580  
Vendas do produto 5: 325
```

Você já percorreu um longo caminho até aqui e aprendeu bastante sobre lógica de programação e Java. A partir de agora, vamos continuar explorando o tema abordando subprogramas.

8. Subprogramas

Até o momento, escrevemos programas cujas instruções estão todas centradas no método (procedimento) **main** de um programa Java. No entanto, reflita que, antes mesmo de conceituar subprogramas (chamados de métodos em Java), você já usou alguns em Java, tais como, **print** e **println**, **next**, **nextLine**, **nextInt** e **nextDouble**.

Vamos imaginar então o método **println**. Você sabia que esse método tem muitas linhas de código para imprimir uma simples mensagem? Vamos mostrar o código da implementação, mas não se preocupe com o conteúdo, apenas observe que, diretamente, são 10 linhas de código:

</> Código na área

```
/* o código abaixo foi extraído da classe System do pacote java.lang da
   Linguagem de Programação Java.
*/
. . .
public void println(String x) {
    if (getClass() == PrintStream.class) {
        writeln(String.valueOf(x));
    } else {
        synchronized (this) {
            print(x);
            newLine();
        }
    }
}
. . .
```

O método ***println*** tem muitas linhas de código, e é aí que vamos começar a buscar uma compreensão de subprogramas. Para iniciar, analise um dos programas feitos até aqui: qualquer um deles tem mais do que uma chamada do ***println***. Se não tivéssemos o recurso de **subprogramas**, teríamos de implementar o código anterior em todas as ocorrências em que se deseja imprimir qualquer mensagem na tela do usuário. Esse é um dos motivos que nos leva a pensar na necessidade de implementação de subprogramas e fazer códigos que possam ser reutilizados em várias partes do programa.

Subprograma: é uma parte autônoma e independente de um programa de computador que realiza uma tarefa específica. É uma unidade de código que pode ser **chamada** e executada a partir de outras partes do programa principal.

Como já mencionamos, basicamente escrevemos todos os programas até agora dentro do bloco de instruções do método (função) ***main***. De certa forma, construímos o programa seguindo uma lógica de programação que normalmente segue a ordem **Entrada de Dados, Processo** e, por fim, mas não menos importante, as Saídas esperadas. A figura a seguir representa essa situação:

Entrada de Dados, Processo e Saídas

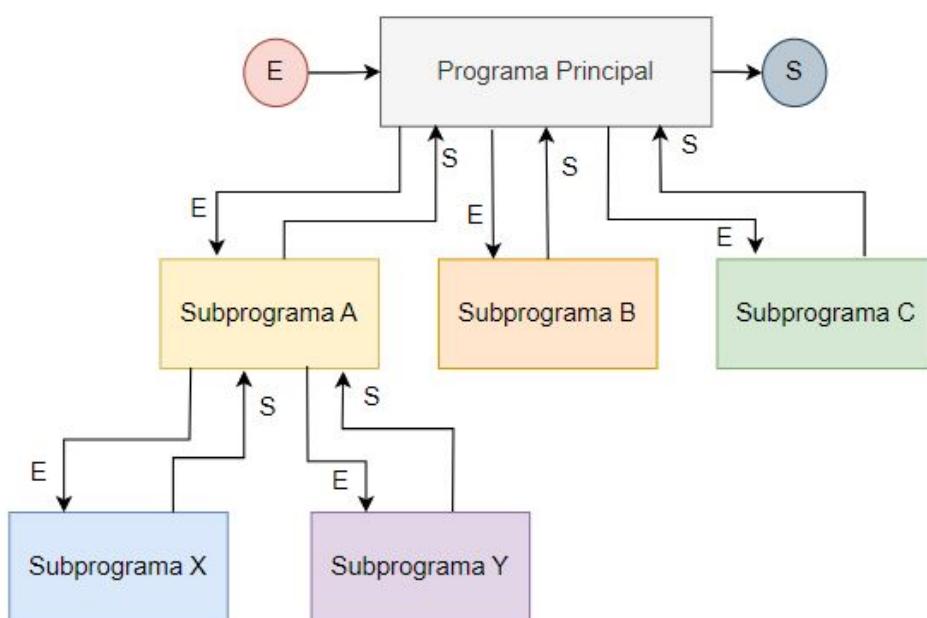


Fonte: Elaborada pelos autores (2024).

No entanto, quando introduzimos os subprogramas, dividimos o conteúdo do programa principal em blocos de instruções menores, cada um responsável por uma ação que, somadas com as demais, vai dar o mesmo sentido que esperávamos do programa principal. O programa principal, neste caso, se comportará como um maestro que inicia o programa, chama os subprogramas de que necessita para executar as tarefas necessárias e encerra a execução do programa.

A imagem a seguir representa o Programa Principal coordenando à execução do programa realizando as chamadas necessárias. Note que subprogramas podem requerer ações de outros subprogramas para executarem suas tarefas também. As chamadas de subprogramas podem ser entendidas como programas menores, que recebem **Entradas** e retornam resultados, as **Saídas**.

Programa Principal



Fonte: Elaborada pelos autores (2024).

Os subprogramas são utilizados para modularizar e organizar o código, tornando-o mais legível, reutilizável e fácil de manter. Eles permitem dividir um programa em unidades lógicas e independentes, cada uma responsável por uma parte específica da lógica de negócio. Isso facilita a compreensão, o teste e a manutenção do código, além de promover a reutilização de código em diferentes partes do programa. Uma vez implementado, o subprograma deve ser chamado em alguma parte do programa para que sua tarefa seja executada. Deste modo, o programa chamador aguarda enquanto as instruções do subprograma são executadas. Quando o subprograma é finalizado, o controle é devolvido para o programa chamador.

Em programação, os subprogramas podem ser de dois tipos principais:



Procedimentos



Funções

Um procedimento é um subprograma que executa uma sequência de instruções para realizar uma determinada tarefa. Ele tem como principal característica não retornar nenhum valor específico para quem o chamou.

Uma função é um subprograma semelhante a um procedimento, mas que retorna um valor específico para quem a chamou após sua execução. Esse valor pode ser utilizado em outras partes do programa.

Embora as designações PROCEDIMENTOS e FUNÇÕES sejam comuns no estudo de lógica de programação, algumas linguagens de programação adotam outras denominações. Por exemplo, em C, tudo é chamado de função (função que retorna valor - FUNÇÃO, e função que não retorna valor - PROCEDIMENTO), e em Java, tudo é chamado de método (método que retorna - FUNÇÃO, ou não retorna valor - PROCEDIMENTO).

Analogamente falando, uma função ou um procedimento é como uma fábrica de bolos. Há rotinas para fazer o bolo em si, mas muitas outras são necessárias para que o bolo saia quentinho do forno. Por exemplo, o processo de produção de farinha, manteiga, chocolate e outros.



Muitas vezes, não nos preocupamos com esses detalhes, mas, para o processo de fazer o bolo, os outros processos também são necessários. Nesse sistema se dá o trigo de Entrada, temos a farinha de Saída. Dá-se cacau e açúcar de Entrada, tem-se o chocolate na Saída. Dá-se leite, sal e outros ingredientes de Entrada, tem-se a manteiga de Saída. Todas essas Saídas são utilizadas de entrada no processo de fabricação do bolo, e após ser executado, temos a Saída que será o próprio bolo.

A sintaxe na linguagem de programação Java para declarar e implementar métodos (função ou procedimento) em Java é mostrada a seguir. **Todo método em Java deve ser implementado dentro do escopo de uma classe.**

</> Código na área

```
[modificadorDeAcesso] tipoDeRetornoSaida nomeDoMetodo(parametrosEntrada) {  
    // Corpo do método  
    // Código para realizar a tarefa  
    // Pode incluir declarações de variáveis, estruturas de controle,  
    //     chamadas de outros métodos, etc.  
    return valorDeRetorno; // opcional, apenas se o método tem um tipo de  
    // retorno diferente de void  
}
```

Na sintaxe apresentada, cada elemento tem o seguinte significado:

modificadorDeAcesso

Indica o nível de acesso do método (*public*, *private*, *protected* ou *default*). Inicialmente, vamos adotar a declaração *public*, já que esse conceito é tratado com mais profundidade na disciplina de Programação Orientada a Objetos.

tipoDeRetornoSaida

Indica o tipo de dado que o método retorna. Se o método não retorna nada, o tipo de retorno é *void* e, neste caso, o método é considerado um procedimento. Caso seja diferente de *void* (por exemplo: *int*, *double*, *String*, ...), o método pode ser considerado uma função que faz um processo, gera um resultado e este é retornado (saída) para quem chamou o método.

nomeDoMétodo

O nome do método deve seguir as convenções de nomenclatura em Java. Normalmente, se trata de uma ação, remete a um processo, por isso, comumente é nomeado com um verbo no imperativo (calcular, somar, verificar, calcularImposto). É por esse nome que o método será chamado em outros locais no programa (como é chamado o método **println** por exemplo).

parametrosEntrada

São os parâmetros que o método recebe (se houver). Eles são separados por vírgulas e seguem o formato **tipo nome**.

Corpo do método

O conjunto de instruções que compõem o método definem a lógica dele. No corpo do método, é possível escrever qualquer instrução da linguagem Java (**declaração de variáveis, if, while, for, chamada de outros métodos**).

return

A palavra-chave **return** é usada para retornar um valor do método. O comando **return** pode ser omitido caso o **tipoDeRetornoSaída** seja declarado como **void**; do contrário, passa a ser obrigatório.

valorDeRetorno

É o valor que o método retorna (saída), ou seja, o resultado gerado pelo processamento do método. Esse é o valor que será retornado ao chamador do método.

A partir da sintaxe a seguir, são apresentados alguns exemplos de métodos:

</> Código na área

```
public class ProcedimentoExemplo {  
    public static void imprimirMensagem() {  
        System.out.println("Olá, mundo!");  
    }  
  
    public static void main(String[] args) {  
        imprimirMensagem();  
    }  
}
```

No primeiro exemplo, a classe **ProcedimentoExemplo**, é possível observar o método **imprimirMensagem**. Da forma como foi implementado, ele é caracterizado como **procedimento**. O programa conta a implementação de dois métodos, o **main** e **imprimirMensagem**, ambos implementados dentro do escopo da classe **ProcedimentoExemplo**. O programa começa a ser executado pelo método **main**. O método **main** tem uma única instrução, que é a chamada do método **imprimirMensagem**. O método **imprimirMensagem** não recebe qualquer parâmetro de entrada e não produz nenhum resultado, apenas executa a chamada do método **println** para exibir a mensagem “**Olá, mundo!**”, por isso, não há declaração de **parametrosEntrada** e **tipoDeDadosRetornoSaida** é **void**. Ao ser chamado, o método **imprimirMensagem** é executado, enquanto isso, o **main** aguarda que o método chamado seja finalizado, para que ele possa ser também encerrado na sequência, ou seja, o **main** aguarda a finalização do método **imprimirMensagem** para dar continuidade.

Considerando que um programa sempre começa a execução por um programa principal, vale destacar que, em Java, o programa sempre começará pelo método main. A partir daí ocorrem as chamadas de outros métodos até que o programa finalize a execução do método main. Assim, é o método main que inicia a execução do programa e é por ele também que o programa finaliza. Em Java, toda função e todo procedimento são chamados de Método.

No código seguinte, é possível observar um método (com a característica de função) com passagem de parâmetro e retorno de valor:

</> Código na área

```
public class FuncaoExemplo {  
    public static int somar(int a, int b) {  
        return a + b;  
    }  
  
    public static void main(String[] args) {  
        int resultado = somar(3, 5);  
        System.out.println("Resultado da soma: " + resultado);  
    }  
}
```

No programa **FuncaoExemplo** (código acima) **somar** é um método que recebe dois parâmetros de entrada, as variáveis inteiras **a** e **b**. As variáveis são utilizadas para executar um processo que é a operação de soma. Essa operação resulta em um número que é retornado para quem chamou o método. No exemplo, o programa começa a ser executado pelo método **main**. A primeira linha tem duas instruções. Primeiro, a chamada do método **somar** e, depois, a **atribuição** para a **variável resultado**. Na chamada do método **somar** (instrução a direita da atribuição “=”), são levados como entrada os valores **3** e **5**, que serão colocados respectivamente nas variáveis de entrada **a** e **b** do método **somar(int a, int b)**. A soma será processada, e o resultado gerado é devolvido para o método **main** (chamador), que finaliza a instrução atribuindo o valor para a variável **resultado**. A próxima e última instrução do método **main** é usar a variável **resultado** para imprimir o valor encontrado na tela do usuário.

Um método também pode receber arrays como parâmetro de entrada e até mesmo retornar um array. O programa a seguir mostra essa implementação.

</> Código na área

```
import java.util.Random;

public class MetodosComArray {
    /**
     * Método main - por onde começa a execução do programa
     * @param args
     */
    public static void main(String[] args) {
        int[] numerosGerados = new int[10];
        numerosGerados = gerarNumeros();
        imprimirNumeros(numerosGerados);
    }

    /**
     * Método para gerar 10 números aleatórios de 1 a 100 e
     * retorna na forma de vetor de inteiros (int[])
     * @return um vetor de inteiros
     */
    public static int[] gerarNumeros() {
        int[] numeros = new int[10];
        for (int i = 0; i < numeros.length; i++) {
            int num = gerarUmNumero();
            numeros[i] = num;
        }
        return numeros;
    }

    /**
     * Método que gera um número aleatório e retorno para o chamador
     * @return um valor inteiro
     */
    public static int gerarUmNumero() {
        Random r = new Random();
        int numero = r.nextInt(60) + 1;
        return numero;
    }

    /**
     * Método que recebe um vetor de inteiros e
     * imprime o seu conteúdo
     * @param numeros um vetor de inteiros
     */
    public static void imprimirNumeros(int[] numeros) {
        for(int i = 0; i < numeros.length; i++) {
            System.out.println("Número " + (i+1) + ": " + numeros[i]);
        }
    }
}
```

Agora, você já está preparado para colocar esse conteúdo em prática. Lembre-se de que, sempre que tiver alguma dúvida, poderá contar com os tutores. Bons estudos!

Finalizando

A conclusão desta Unidade Curricular marca o término de uma jornada em que foram abordados tanto os fundamentos da lógica de programação quanto os conceitos essenciais da linguagem de programação Java. Desde a exploração das diferentes formas de se escrever um algoritmo, como fluxogramas, narração descritiva e pseudocódigo, até a compreensão das instruções de entrada e saída de dados, variáveis, estruturas de controle condicionais, estruturas de repetição, vetores, matrizes e subprogramas, cada capítulo ofereceu uma base sólida para iniciantes e uma revisão útil para programadores mais experientes.

Agora, munido com conhecimentos valiosos e habilidades práticas, você está preparado para mergulhar mais profundamente no mundo da programação, explorando aplicações mais avançadas e desafios complexos, que serão abordados nas demais disciplinas ao longo deste curso. Esperamos que esse conteúdo seja apenas o começo de uma jornada emocionante e gratificante no mundo da programação de computadores.

Referências

ASCII. *In:* WIKIPÉDIA, a enclopédia livre. Flórida: Wikimedia Foundation, 2024a. Disponível em: <https://pt.wikipedia.org/w/index.php?title=ASCII&oldid=67928440>. Acesso em: 19 jun. 2024.

ALGORITMO. *In:* WIKIPÉDIA, a enclopédia livre. Flórida: Wikimedia Foundation, 2024b. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Algoritmo&oldid=67808821>. Acesso em: 19 jun. 2024.

FERREIRA, A. B. H. **Aurélio século XXI**: o dicionário da Língua portuguesa. 3. ed. rev. e ampl. Rio de Janeiro: Nova Fronteira, 1999.

FORBELLONE, A.; EBERSPÄCHER, H. **Lógica de programação**: a construção de algoritmos e estrutura de dados. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

FLUXOGRAMA. *In:* WIKIPÉDIA, a enclopédia livre. Flórida: Wikimedia Foundation, 2022. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Fluxograma&oldid=64129194>. Acesso em: 19 jun. 2024.

KNUTH, D. E. **The Art of Computer Programming, Volume 1: Fundamental Algorithms**. 3. ed. Boston: Addison-Wesley, 1997.

MOURA, L. Introdução à Linguagem de Programação Java. **Dio.**, 2024. Disponível em: <https://www.dio.me/articles/introducao-a-linguagem-de-programacao-javascript>. Acesso em: 19 jun. 2024.

PESSOTO, C. Há vagas: setor de TI de SC projeta 100 mil postos de trabalho até 2025. **SC Inova**, 2023. Disponível em: <https://scinova.com.br/setor-de-ti-de-sc-projeta-100-mil-postos-de-trabalho-ate-2025/>. Acesso em: 19 jun. 2024.

PSEUDOCÓDIGO. *In:* WIKIPÉDIA, a enclopédia livre. Flórida: Wikimedia Foundation, 2023. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Pseudoc%C3%B3digo&oldid=65249211>. Acesso em: 19 jun. 2024.

TREMBLAY, J. P.; SORENSEN, P. G. **An Introduction to Data Structures with Applications**. 2. ed. New York: McGraw-Hill, 1984

