

# LÓGICA DE PROGRAMAÇÃO PARA INICIANTEs



Gustavo Furtado de Oliveira Alves

Olá nobre amigo(a)! Que bom ter você aqui comigo lendo este ebook de lógica de programação para iniciantes.

Meu nome é **Gustavo Furtado de Oliveira Alves**, sou Mestre em computação aplicada pelo Instituto Nacional de Pesquisas Espaciais (INPE), Engenheiro da Computação pela ETEP Faculdades e Técnico em Informática pela Escola Técnica Pandiá Calógeras. Possuo as certificações ASF, SCWCD e SCJP e trabalho com desenvolvimento de softwares desde 2007. Criei este ebook com o intuito de te ajudar a criar uma base sólida de conhecimento que te permitirá criar programas em qualquer linguagem de programação que o mercado de trabalho requisitar. Ao longo deste ebook, você aprenderá os conceitos básicos por trás da programação e será capaz de criar pequenos programas usando lógica de programação.



A seguir vamos ver o que você vai aprender em cada capítulo deste ebook:

## Capítulo 1: Por que aprender programação?

- Porquê você PRECISA aprender programação.
- Preciso saber inglês pra aprender programação?
- Qual linguagem você deve escolher para aprender lógica de programação.

## Capítulo 2: Criando os seus primeiros programas

- O que é um Algoritmo.
- Como instalar o Visualg para aprender lógica de programação.
- Mão na massa: criando os seus primeiros programas.

## Capítulo 3: Variáveis, constantes e tipos de dados

- Como armazenar dados na memória do computador.
- A diferença entre variáveis e constantes.
- O que são e como usar os tipos de dados.

- A diferença entre os tipos de dados primitivos e os tipos de dados customizados.

### **Capítulo 4: Operadores**

- Os 3 tipos de operadores: Aritméticos, Lógicos e Relacionais.

### **Capítulo 5: Tomando decisões!**

- Fazendo seu software tomar decisão com SE-ENTÃO-SENÃO.

### **Capítulo 6: Tomando decisões entre muitas opções**

- A estrutura de controle de fluxo ESCOLHA-CASO.
- A diferença de um HUB e um SWITCH.

### **Capítulo 7: Loops Básicos!**

- Entendendo estruturas de repetição de uma vez por todas
- ENQUANTO-FAÇA
- REPITA-ATÉ

### **Capítulo 8: Loops Pré-definidos**

- a estrutura de repetição mais usada: PARA-FAÇA!.

### **Capítulo 9: Vetores e Matrizes.**

- O tipo de dados customizado mais básico da computação: array

### **Capítulo 10: Funções e Procedimentos.**

- O que são e como criar as suas próprias funções e procedimentos.

### **Resolução dos Exercícios**

# Capítulo 1 - Por que aprender programação?

"I think everybody in this country should learn how to program a computer because it teaches you how to think."

— Steve Jobs, the Lost Interview

**Tradução da frase:** *"Eu acho que todos neste país deveriam aprender como programar um computador porque isto te ensina como pensar."*

Assim como Steve Jobs disse a frase acima, eu também penso que todos deveriam aprender a programar.

Como podemos perceber nos últimos tempos a evolução tecnológica melhora a vida das pessoas e o Software é um dos pilares da tecnologia do nosso tempo. Com softwares pode-se resolver muitos problemas do dia-a-dia. Então por que não aprender a criar softwares para automatizar tarefas e assim economizar tempo e dinheiro?

## 4 Motivos para você começar a aprender programação AGORA!

### 1º Motivo: Você vai mudar a sua forma de pensar!

Aprender programação ajuda muito a forma como as pessoas pensam, principalmente por desenvolver a disciplina da **lógica**, que é um campo da filosofia criado por Aristóteles que cuida das regras do bem pensar, ou do pensar correto, sendo portanto, um instrumento do pensar. A lógica guia o raciocínio humano através de argumentos para chegar a conclusões de verdade.

Quando uma pessoa aprende programar, ela desenvolve uma nova forma de pensar. Todas as decisões passam a ser tomadas levando em consideração pensamentos sistêmicos e racionais que convergem para o bom-senso da razão humana.

Na minha opinião, quando programação for uma disciplina básica da educação, o mundo entrará na próxima era da evolução da humanidade.

### 2º Motivo: Programação é a nova disciplina básica da alfabetização

Muitas pessoas pensam que programação é só pra quem fica o dia inteiro na frente do computador e tem facilidade para mexer na máquina. Isso é natural, principalmente entre as pessoas mais velhas que cresceram sem o contato com a tecnologia atual.

Apesar de parecer intimidante no início, programar não é tão complicado como muitos pensam.

Na verdade, da mesma forma como ler, escrever e fazer cálculos básicos, programação é a nova disciplina básica para alfabetização. Tenho certeza que em poucos anos, programação será ensinado nas escolas de ensino fundamental.



### 3º Motivo: Os softwares estão em tudo!

Já parou pra pensar na quantidade de coisas que estamos direta e indiretamente em contato e que são controladas por softwares? Calculadoras, computadores, celulares, smartphones, tablets, internet, TVs, micro-ondas, geladeiras, caixa eletrônicos, linhas de produção, satélites, carros, impressoras, letreiros digitais, drones, câmeras, semáforos, microcontroladores, pendrive e mais um monte de coisa que eu ficaria horas, talvez dias, enumerando.



Hoje em dia muitas coisas a nossa volta tem uma espécie de vida própria. E tudo é controlado por ... **Softwares!** Dispositivos que são controlados por softwares estão por toda parte, em todos os seguimentos industriais. Agricultura, manufatura, logística, marketing, agropecuária, medicina, etc. Atividades onde você menos imagina têm software!

Isso tudo falando só do presente, deixo a sua imaginação livre para pensar no que vem pela frente nos próximos anos.

Os softwares controlam as máquinas, mas somos nós que criamos os softwares. No futuro próximo, saber criar softwares será como saber dirigir hoje em dia. Você precisa estar preparado para participar dessa nova era!

### 4º Motivo: Você já sabe programar! Só não te contaram...

Diferente do que muitos pensam, programação não é um monte de código que poucos conseguem entender. No fundo, programar é ensinar uma máquina a resolver problemas. Sem perceber, todos nós resolvemos problemas e tomamos decisões o tempo todo!

Para programar, você precisa organizar e entender a forma como você resolve os problemas. Você já toma decisões e realiza tarefas o tempo todo, programar nada mais é do que ensinar uma máquina a fazer isso.

Programação é uma forma de automatizar decisões e atividades através de instruções que um equipamento eletrônico pode seguir para executar uma tarefa autonomamente. Ao entender este conceito, códigos que você achava que eram coisas de outro mundo começam a fazer sentido.

### Preciso saber inglês para aprender programação?

Eu acredito que este seja o maior predador de iniciantes em programação.

É verdade que é possível aprender programação sem saber inglês, eu mesmo sou prova disso.

Quando ingressei no ensino médio e técnico em informática na ETPC (uma escola técnica em Volta Redonda-RJ) eu só sabia o inglês que me foi ensinado no ensino fundamental de uma escola pública, ou seja, quase nada. Importante ressaltar que não desmereço em momento algum os professores de escola pública, são guerreiros, mas todos sabemos como é o sistema público brasileiro de educação. Não é mesmo?

Lembro das minhas primeiras aulas de programação, como era difícil entender o significado dos comandos que o professor ensinava em sala de aula, juro que





sentia dificuldade para decorar palavras básicas em inglês como IF, THEN, ELSE, WHILE, FOR, REPEAT, UNTIL, BEGIN, etc. Bom, acho que deu para perceber que eu não sabia nada mesmo de inglês. Ah! Não tenho um pingão de vergonha disso! Foi só um pequeno obstáculo que estava no meu caminho. Mas já ultrapassado. E se você não sabe inglês, não tenha medo! Basta querer e se esforçar.

Por ser tão importante, quero te dizer isso logo no início deste ebook: **Sim! É possível começar a aprender programação sem saber inglês!** Aliás neste ebook os códigos vão ser todos em português, mas **é muito importante, muito importante mesmo, aprender inglês** para ser um bom programador. Como seu tutor neste início da sua jornada, tenho o dever de te falar isso.

As linguagens de programação profissionais da atualidade são todas em inglês, você também precisará pesquisar na internet para evoluir e resolver problemas dos seus programas, e acredite, a maioria das respostas para as suas dúvidas estarão na sua cara, mas em inglês. Já passei por isso!

Inglês não é um impeditivo para aprender a programar, mas certamente será uma pedra no seu sapato ao longo da sua carreira se você ainda não souber.

Mas não se preocupe, você pode aprender inglês junto com programação. Portanto, **comece já os estudos de inglês em paralelo com a programação.**

Eu escrevi um post sobre isso no blog { **Dicas de Programação** }.

Dê uma olhada depois.

[Quer ser programador? Aprenda inglês!](#)

### Com qual linguagem começar?

A pergunta que é feita por todo mundo que está aprendendo a programar é: Que linguagem de programação devo aprender? resposta é óbvia: **Uma linguagem de programação para iniciantes!**

Você deve ter consciência que ao longo da sua carreira como programador, você sempre terá que aprender uma linguagem de programação nova. Pois todas tem suas vantagens e desvantagens.



Uma dica interessante é: não seja fanático por uma linguagem de programação específica! Muitas pessoas defendem com unhas e dentes a linguagem de programação que têm mais afinidade. Basta olhar nos fóruns da internet as respostas para a pergunta: "Qual a melhor linguagem de programação para iniciantes?"

Mas isso não é bom! **Para cada projeto, cada trabalho, uma linguagem é mais indicada ou não.** Você deve decidir qual usar não com base no seu gosto pessoal, mas nas vantagens que a linguagem oferece para o software que você pretende desenvolver.

Se você é iniciante e não sabe nada de programação, procure uma linguagem de programação que te ajude a aprender lógica de programação.

Só depois de aprender o básico que você deve aprender uma outra linguagem mais profissional.

Muitas pessoas já começam aprendendo lógica de programação com linguagens como Java, C, Python, C#, etc. É totalmente válido e muitos conseguem de fato aprender (especialmente se já souber inglês), mas algumas pessoas podem ter dificuldade de assimilar os conceitos básicos utilizando essas linguagens para começar no mundo da programação.

Então, para quem nunca programou antes, indico fortemente que comece com uma linguagem que o ajude aprender **lógica de programação**. Aprendeu lógica de programação? Agora você está livre pra voar no mundo do desenvolvimento de softwares qual a linguagem que você quiser!

Agora vou te confessar uma coisa. Foi difícil eu entender isso quando comecei a ensinar programação, queria logo que os alunos aprendessem Java, PHP, C, C#, Python, etc. Mas a dificuldade dos iniciantes era conseguir aprender lógica de programação tendo que "decorar" os comandos exigidos pela linguagem.

Por exemplo, para fazer um programinha "Hello World" em JAVA é preciso criar uma classe, um método, e já exigir que o estudante de programação escreva palavras específicas de java como **class**, **public**, **static**, **void** e **main**.

Nesse ponto aliás, Python é uma linguagem muito boa para se aprender, pois vai direto ao ponto. Mas é em inglês.

Não quero aqui entrar em discussão de linguagem X é melhor pra aprender que a linguagem Y, pois há muita discussão sobre isso na internet, principalmente entre pessoas que defendem a "linguagem preferida" ou que argumentam apenas qual é melhor para o mercado de trabalho atual.



O que eu quero deixar claro pra você é que **você deve escolher a linguagem a se aprender baseado no seu objetivo.**

Se o seu objetivo agora é aprender lógica de programação, começar no mundo do desenvolvimento de softwares, utilize uma linguagem que vai te ajudar a aprender lógica de programação! Pronto.

Ou se você já domina lógica e quer aprender uma linguagem pra arrumar um emprego, pesquise no mercado a linguagem que está sendo mais pedida nas vagas de emprego atualmente e vá fundo nos estudos!

Se você quer criar um software embarcado (para foguetes, sondas, satélites, mísseis, etc.), procure uma linguagem apropriada e mais usada para softwares embarcados.

O mercado muda, a linguagem "da moda" hoje pode não estar tão em alta amanhã. Pense nisso. Por isso sempre que você decidir aprender uma linguagem nova de programação, pense no seu objetivo de curto, médio e longo prazo.

Entendeu?

### Paradigma "Como" fazer e paradigma "O que" fazer

Há uma outra discussão muito interessante também sobre aprender ou não lógica de programação no início da carreira, pois algumas linguagens de programação foram criadas para outros paradigmas de programação em que a forma de programar é completamente diferente.

Linguagens como **Lisp, Prolog, IPL**, etc. utilizam paradigmas de programação que não trabalham a forma de "como" fazer e sim "o que" fazer.

Neste ponto, mantenho a minha opinião anterior. Se para atingir o seu objetivo atual, você tenha que aprender alguma dessas linguagens agora, talvez o melhor realmente não seja aprender lógica de programação neste momento.

A decisão do que aprender deve ser sua, baseada no objetivo que você quer alcançar.

Se você quer aprender programação para conseguir um emprego, sugiro que aprenda lógica de programação, pois a maioria das vagas de emprego atuais requerem alguma linguagem de programação do paradigma do "como" fazer, ou seja, implementar algoritmos.

**Estou dedicando este ebook para pessoas que nunca tiveram contato com lógica de programação** e pelas minhas pesquisas, descobri que muitos

não dominam inglês, vamos escrever códigos em português e utilizar uma linguagem simples para transmitir os conceitos importantes, pois o objetivo é ensinar o básico.

Mas como já disse, **é muito importante aprender inglês** se você pretende se profissionalizar em programação.

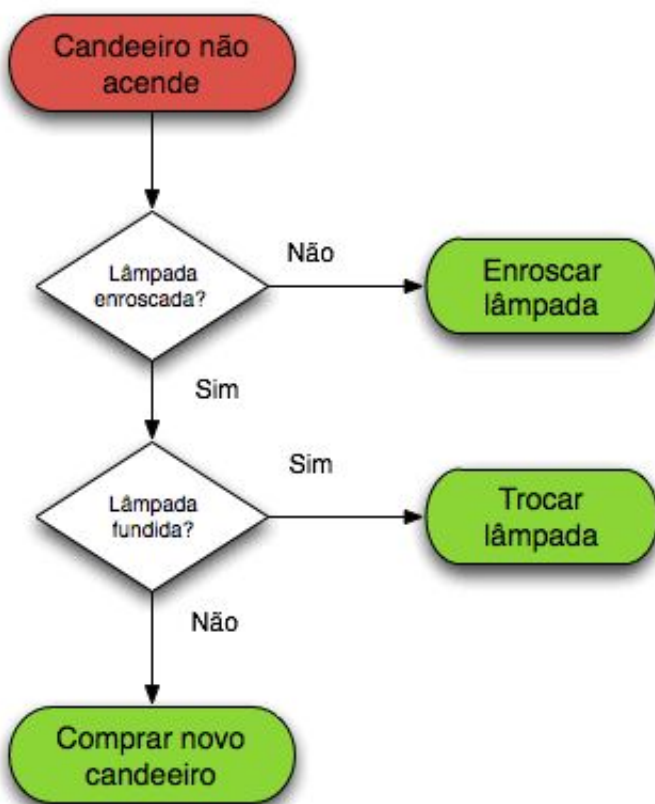
Se você já domina inglês e quiser implementar os exemplos e exercícios desse ebook em outra linguagem mais profissional, ÓTIMO! Fique a vontade, escolha a linguagem que você quer aprender (baseado no seu objetivo) e mãos à obra!

## Capítulo 2 - Criando os seus primeiros programas.

Agora que você já sabe que programação é simplesmente ensinar uma máquina a executar tarefas e tomar decisões, é hora de começar a aprender como fazer isso criando os seus primeiros algoritmos.

### Mas afinal, que é um Algoritmo?

O primeiro passo para se aprender programação não envolve computador, envolve educar a sua mente a explicar em detalhes os passos necessários para executar uma determinada tarefa.



Você deve aprender a modelar um roteiro que explica quando tomar decisões e quando realizar determinadas tarefas, esse roteiro é chamado de "algoritmo".

Você sabia que os primeiros processadores só sabiam realizar somas?

A partir dessa operação básica que o computador sabia fazer, você já imagina um algoritmo para fazer multiplicações?

Talvez você ainda não saiba exatamente como é esse algoritmo, mas com certeza já imaginou que precisa fazer repetidas somas. Certo?

É assim que nós aprendemos fazer multiplicação na escola. E essa também é uma forma de ensinar

uma máquina a fazer multiplicação.

Agora que você já pensou como faz multiplicação através de somas, vou mostrar como eu faria um algoritmo para realizar uma multiplicação de dois números.

### Algoritmo para fazer multiplicação

Algoritmo **Multiplicação de números positivos**

**Declaração de variáveis**

numero1, numero2, resultado, contador: Inteiro

**Início**

leia(numero1)

leia(numero2)

resultado <- 0

contador <- 0

**Enquanto** contador < numero2 **Faça**

    resultado <- resultado + numero1

    contador <- contador + 1

**Fim-Enquanto**

escreva(resultado)

**Fim**

Lendo este algoritmo você pode ter algumas dúvidas na sua cabeça ...

- OK, **Início** e **Fim** eu entendi, mas que raios é **declaração de variável**, **Enquanto**, **Fim-Enquanto**, contador, etc...

Talvez este algoritmo possa ser um pouco complicado para você entender agora, sendo o primeiro algoritmo que te mostro. Mas não se preocupe em entender cada passo do algoritmo agora. Continue lendo que vou te mostrar alguns algoritmos mais simples neste capítulo. Ao final você entenderá exatamente como esse algoritmo funciona... Mas antes vamos ver a ferramenta que vamos usar ao longo deste ebook.

### A melhor ferramenta para aprender lógica de programação

Sabe qual a melhor ferramenta de estudos para aprender lógica de programação?

**Caderno, lápis e borracha!** Sim, essa é a melhor ferramenta para aprender lógica de programação!



No começo, usa-se mais a borracha do que o lápis! rs.

Você deve estar se perguntando, com tantos aplicativos hoje em dia eu não posso usar o meu Smartphone pra aprender algoritmo? Bom, acredito que hoje com tantas distrações na internet talvez seja realmente melhor se desligar disso tudo para conseguir aprender algo. O bom e velho conjunto de lápis e caderno nos força a nos desligarmos um pouco. O que acha? Mas fique a vontade para utilizar o seu software editor de texto predileto! ;)

Uma técnica que gosto muito para me ajudar na concentração e ter mais produtividade é a **Técnica Pomodoro**.

Não é o foco deste ebook, mas eu escrevi um artigo sobre essa técnica no blog { **Dicas de Programação** }. Se quiser saber mais clique no link abaixo:

[Clique AQUI para conhecer a técnica pomodoro!](#)

Voltando ao assunto, se não vai usar lápis e borracha e quiser utilizar um software para te ajudar a aprender programação. Sugiro que você utilize o Visualg, pois neste ebook vamos utilizá-lo para escrever e compilar códigos em português.

Na minha opinião o VisuAlg é a melhor IDE (Ambiente de desenvolvimento) para iniciantes em programação implementarem seus algoritmos.

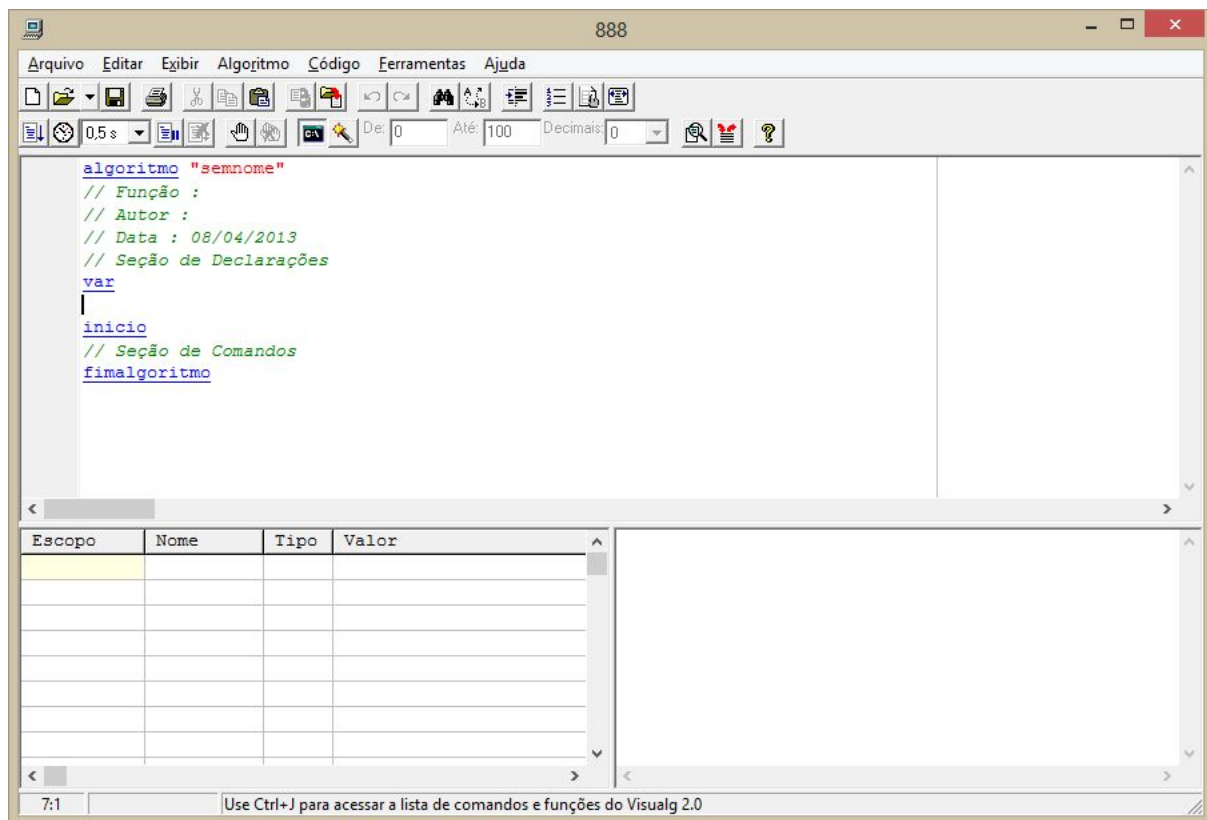
O Visualg foi criado por um brasileiro (Cláudio Morgado de Souza), é fácil de ser usado e compila pseudo-códigos escritos em português, também conhecidos como "Portugol".

## Instalando o Visualg no seu computador

A instalação do VisuAlg é muito simples e o software pode ser instalado em Windows 95 ou posterior. Para instalar o Visualg basta baixá-lo através do link abaixo, extrair o executável e rodar. Pronto.

[Clique AQUI para fazer o download do Visualg](#)

Obs: Quem usa linux, o Visualg funciona perfeitamente no Wine.



A tela do VisuAlg compõe-se da barra de menu, barra de tarefas, barra de ferramentas, do editor de textos (que toma toda a sua metade superior), do quadro de variáveis (no lado esquerdo da metade inferior), do simulador de saída (no correspondente lado direito) e da barra de status. O programa já inicia com o "esqueleto" de um algoritmo. Como você pode ver na figura acima.

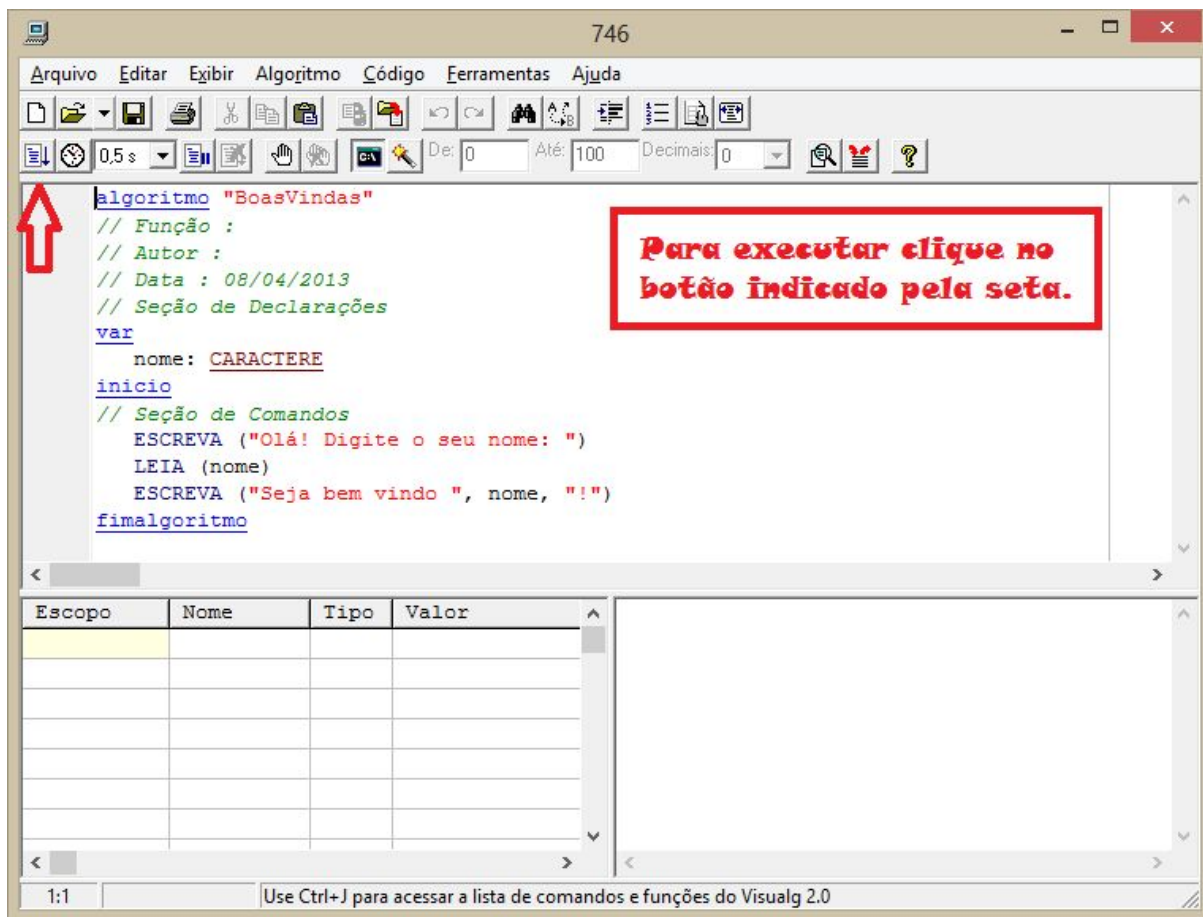
O professor **Antonio Carlos Nicolodi** reformulou o Visualg e lançou a versão 3.0 com uma interface nova e algumas melhorias. Para fazer os exercícios deste e-book você pode usar tanto a versão 3.0 quanto a 2.5. Como você preferir.

**[Você pode baixar qualquer uma das duas versões clicando AQUI!](#)**

## Criando o seu primeiro programa!

Agora que você já tem o VisuAlg, é hora de criar o seu primeiro programa. O famoso "Hello World". Abra o visualg e escreva o algoritmo abaixo:





Vamos entender esse primeiro programa que você criou.

1º Na primeira linha, nós colocamos o nome do algoritmo **"BoasVindas"**.

2º As quatro linhas seguintes são comentários, ou seja, é ignorado pelo compilador, não é um comando de algoritmo. Toda linguagem de programação tem alguma forma de fazer comentários no código. No Visualg os comentários começam com duas barras. Assim: //

**Nota para o iniciante:** Embora os comentários não sejam interpretados como comandos na hora de executar o programa, eles são muito importantes quando se escreve softwares, pois através dos comentários a gente explica o que uma parte do código faz para um outro programador que trabalhará neste mesmo código no futuro. Lembre-se: este programador pode ser você! É uma boa prática comentar códigos.

3º Em seguida vemos as declarações de variáveis. Nós declaramos uma variável chamada **nome** do tipo **CARACTERE**. No próximo capítulo eu vou explicar o que

é uma variável, mas por agora só entenda que nós podemos armazenar valores em uma variável.

4º O programa começa de fato após a cláusula **inicio**. Perceba que depois do início tem outro comentário.

5º A primeira coisa que fazemos no programa é escrever na tela para o usuário: **Olá! digite o seu nome:**

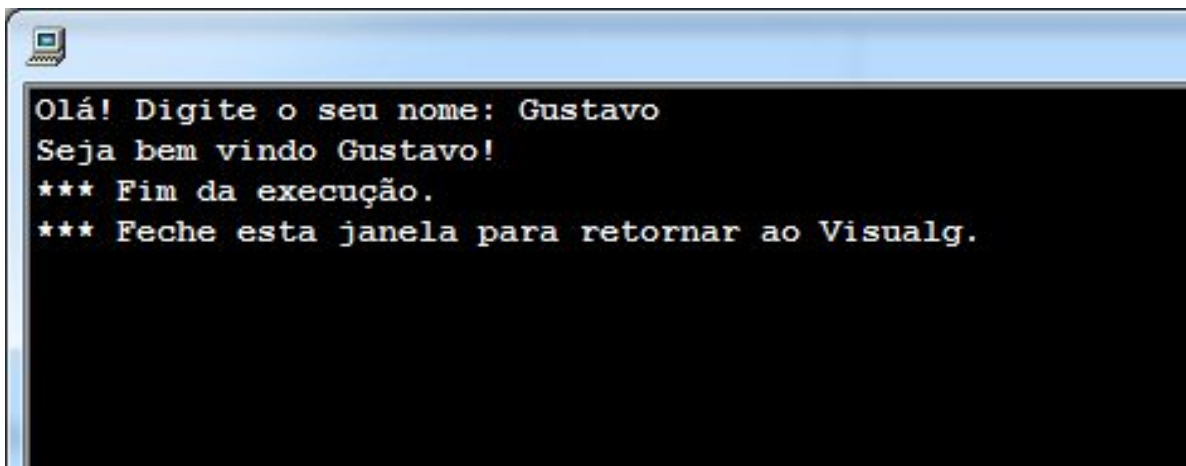
Nós fizemos isso através da função **ESCREVA**. Também falarei sobre as funções mais pra frente neste ebook (capítulo 10), por hora, pense que a função vai fazer alguma coisa pra gente. No caso, escrever um texto na tela.

6º Na linha seguinte, nós capturamos o que o usuário digitou através da função **LEIA**. E armazenamos o texto que o usuário digitou na variável **nome**.

7º Por fim, nós mostramos na tela (novamente através da função **ESCREVA**): **Seja bem vindo (o valor da variável nome)!**

Note que nós juntamos ao texto **Seja bem vindo** o valor da variável **nome**. Se o usuário digitou **José** o programa vai exibir na tela: **Seja bem vindo José!**

Veja na imagem abaixo como acontece a execução do programa que acabamos de criar:



```
Olá! Digite o seu nome: Gustavo
Seja bem vindo Gustavo!
*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

### Entendendo o algoritmo da multiplicação

Agora que você criou o seu primeiro programa, vamos tentar entender aquele algoritmo da Multiplicação que eu falei no começo deste capítulo. mas antes vamos implementá-lo no Visualg.

```
algoritmo "Multiplicação de números positivos"

var
    numero1, numero2, resultado, contador: INTEIRO
inicio

    ESCREVA ("Digite o primeiro número da multiplicação: ")
    LEIA (numero1)
    ESCREVA ("Digite o segundo número da multiplicação: ")
    LEIA (numero2)

    resultado := 0
    contador := 0

    ENQUANTO contador < numero2 FACA
        resultado := resultado + numero1
        contador := contador + 1
    FIMENQUANTO

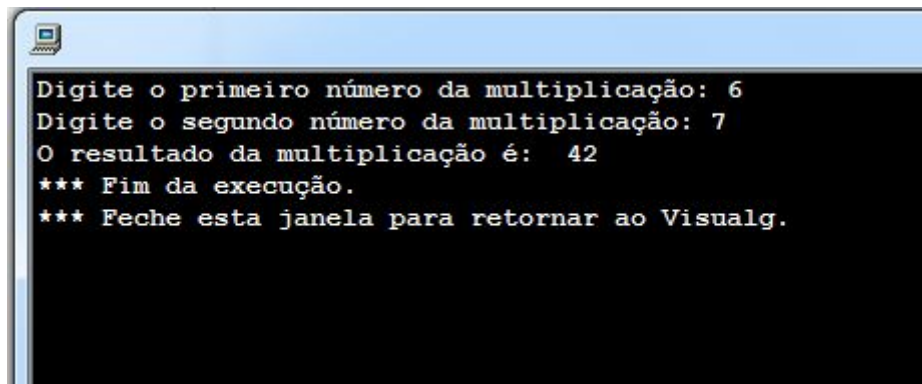
    ESCREVA ("O resultado da multiplicação é: ", resultado)

fimalgoritmo
```

Para entender o algoritmo, é importante definir algumas coisas:

- **Variável** (assunto do próximo capítulo!) é um espaço alocado na memória para armazenar dados. No algoritmo, foram criadas 4 variáveis: **numero1**, **numero2**, **resultado** e **contador**, elas são do tipo inteiro, também veremos isso mais pra frente neste ebook (capítulo 3).
- O símbolo "==" representa uma atribuição de valor a uma variável. Por exemplo, (**resultado := resultado + numero1**) armazena na variável resultado, o valor da própria variável resultado (no momento atual), mais o valor da variável numero1. (Falaremos sobre operadores no capítulo 4)
- O comando **leia(numero1)**, significa que o algoritmo está lendo o que o usuário digita e armazenando na variável **numero1**. (Falaremos sobre funções no capítulo 10)
- O comando **ENQUANTO** é uma estrutura de controle de fluxo do tipo "Estrutura de repetição" (Vamos ver isso no capítulo).

Vamos ver qual seria o resultado da execução deste algoritmo?

A screenshot of a Visualg window with a black background and white text. The text displays a simple multiplication algorithm: it prompts for the first number (6), the second number (7), shows the result (42), and ends with two lines of asterisks indicating the execution is finished and the window should be closed to return to Visualg.

```
Digite o primeiro número da multiplicação: 6
Digite o segundo número da multiplicação: 7
O resultado da multiplicação é: 42
*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

Este algoritmo eu não vou explicar detalhadamente aqui. Prefiro dar as informações que você precisava para compreender. É importante perceber que existem regras e recursos para ensinar uma máquina a executar uma tarefa.

Se você não conseguiu entender este algoritmo da multiplicação, não se preocupe, tem coisa nele que eu ainda vou explicar. Mas já dá pra você ter uma ideia de como um algoritmo funciona, como eu transferi a forma de resolver o problema da multiplicação da maneira que eu sei fazer, porque aprendi quando criança, para o computador resolver sozinho. E é isso que você vai fazer quando estiver programando!

### Exercício para treinar

Inaugurando a sequência de exercícios deste ebook, vou te lançar um desafio!

Quero que você escreva um algoritmo para calcular a média de um aluno através de suas 4 notas no ano letivo.

Média = (nota1 + nota2 + nota3 + nota4) / 4

Ou seja, o algoritmo precisa ler as quatro notas que o usuário digitar, calcular a média e exibir na tela para o usuário.

A resposta para este exercício está no final do e-book, mas é muito importante que você tente fazer este exercício sozinho. Com o que aprendeu até agora, você já é capaz de resolver este exercício.

No próximo capítulo vamos começar a fazer os nossos programas tomarem decisões.

Mas antes, tente praticar criando alguns algoritmos básicos. As dificuldades no primeiro contato com programação é normal, mas neste ebook você vai aprender como superar isso fazendo exercícios.

Para aprender lógica de programação é necessário praticar bastante, então, mãos à obra!

## Capítulo 3 - Variáveis, constantes e tipos de dados.

### O que são variáveis constantes?

Programas de computador utilizam os recursos de hardware mais básicos para executar [algoritmos](#). Enquanto o processador executa os cálculos, a memória é responsável por armazenar dados e servi-los ao processador. O recurso que nós utilizamos em nossos programas para escrever e ler dados da memória do computador é conhecido como **variável**, que é simplesmente **um espaço na**



**memória o qual reservamos e damos um nome.** Por exemplo, podemos criar uma variável chamada "idade" para armazenar a idade de uma pessoa. Você pode imaginar uma variável como uma gaveta "etiquetada" em um armário. Chamamos este espaço alocado na memória de **variável**, porque o valor armazenado neste espaço de memória pode ser alterado ao longo do tempo, ou seja, o valor ali alocado é "variável" ao longo do tempo. Diferente das **constantes**, que é um espaço reservado na memória para armazenar um valor que não muda com o tempo. Por exemplo, o valor PI (3.14159265359...) que nunca vai mudar.

### Como funciona uma variável em um algoritmo

Para não restar dúvidas quanto ao funcionamento de uma variável, vamos ver como elas funcionam em um algoritmo:

Algoritmo "Teste de Variável"

Declaração das variáveis

nome : Texto

Início

nome <- "João"

imprimir(nome)

nome <- "Maria"

imprimir(nome)



Fim

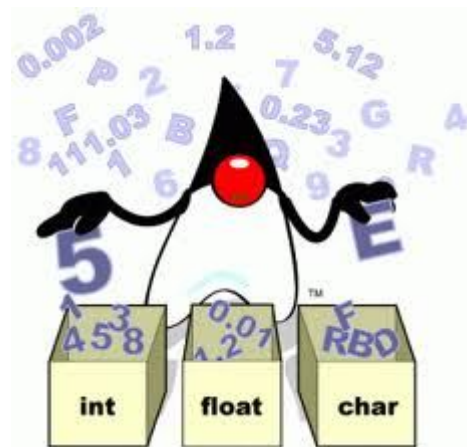
Neste algoritmo, declaramos uma variável chamada *nome* do tipo "Texto". Inicialmente armazenamos o texto "João" na variável *nome*, e mandamos imprimir na tela o valor desta variável. Neste momento aparece na tela o texto "João", em seguida alteramos o valor da variável para "Maria" neste momento o texto "João" é apagado da variável (memória) e em seu lugar é armazenado o texto "Maria". Em seguida, mandamos imprimir na tela novamente o valor da variável, então aparece na tela o texto "Maria".

Deu pra perceber como o valor da variável *nome* pode ser alterado com o tempo?

### Tipos de dados

Para otimizar a utilização da memória, nós definimos um tipo de dados para cada variável. Por exemplo, a variável *nome*, deve armazenar textos, já a variável *idade* deve armazenar apenas números inteiros (sem casa decimal), na variável *sexo* podemos armazenar apenas um caractere ("M" ou "F"). Seria correto armazenarmos o valor "M" na variável *idade*? Não né? Por isso devemos especificar em nossos algoritmos o tipo de cada variável.

Podemos classificar os tipos de dados em basicamente duas categorias, os tipos de dados primitivos e os tipos de dados customizados.



### Tipos de dados primitivos

Existem quatro tipos de dados primitivos, algumas linguagens subdividem estes tipos de dados em outros de acordo com a capacidade de memória necessária para cada variável, mas no geral, os tipos de dados primitivos são:

- **INTEIRO:** Este é o tipo de dados para valores numéricos negativos ou positivos, sem casas decimais. Por exemplo uma variável *idade*.
- **REAL:** Este é o tipo de dados para valores numéricos negativos ou positivos, com casas decimais. Por exemplo uma variável *peso*.

- **LÓGICO:** Este tipo de dados pode assumir apenas dois valores VERDADEIRO ou FALSO. Também é conhecido como **booleano**. Reserva apenas um bit na memória, onde o valor 1 representa VERDADEIRO e o valor 0 representa FALSO. Por exemplo uma variável *status\_da\_lampada* (acesa ou apagada).
- **TEXTO:** Tipo de dados para variáveis que armazenam textos. Por exemplo uma variável *nome*.

Algumas linguagens de programação dividem esses tipos primitivos de acordo com o espaço necessário para os valores daquela variável. Na linguagem Java por exemplo, o tipo de dados inteiro é dividido em 4 tipos primitivos: `byte`, `short`, `int` e `long`. A capacidade de armazenamento de cada um deles é diferente.

- **byte:** é capaz de armazenar valores entre -128 até 127.
- **short:** é capaz de armazenar valores entre - 32768 até 32767.
- **int:** é capaz de armazenar valores entre -2147483648 até 2147483647.
- **long:** é capaz de armazenar valores entre -9223372036854775808 até 9223372036854775807.

Mas essa divisão é uma particularidade da linguagem de programação. O objetivo é otimizar a utilização da memória. Em algumas linguagens de programação não é necessário especificar o tipo de dados da variável, eles são identificados dinamicamente. Porém, é necessário informar o tipo de dados de cada variável em algoritmos.

## Tipos de dados customizados

A partir dos tipos de dados primitivos podemos criar outros tipos de dados utilizando uma combinação de variáveis. São estruturas de dados, classes, vetores, matrizes, etc.

Por exemplo, uma classe chamada *Carro* é um tipo de dados que agrupa outras variáveis básicas como **marca**, **cor**, **ano**, **modelo**, etc.

Um *vetor* é um agrupamento de variáveis do mesmo tipo, uma *matriz* é um agrupamento de vetores. Enfim, a base de todos os tipos de dados são os tipos de dados primitivos, independente da linguagem de programação.

```
class Carro {  
  
    String tipo;  
    String cor;  
    String placa;  
    int numPortas;  
  
    void setCor(String c){  
        cor = c;  
    }  
  
    String getCor(){  
        return cor;  
    }  
}
```

Claro, em **Programação Orientada a Objetos** há todo um conceito para a criação de *classes* que, além de *atributos* também tem *operações*, o estudo de *estruturas de dados* também vai muito além de apenas formar tipos de dados a partir de outros. Mas não se preocupe com isso por agora. Apenas entenda que são tipos de dados formatos a partir de outros tipos de dados.

Diferente dos tipos de dados primitivos que já são implementados internamente pelas linguagens de programação, esses tipos de dados são criados pelo programador. Enfim, neste e-book não vamos utilizar classes. Os tipos de dados customizados que vamos aprender serão vetores e matrizes, assuntos do capítulo 9.

Espero que você tenha entendido esses dois assuntos, pois saber como funcionam as variáveis/constantes e os tipos de dados é de suma importância para você se tornar um bom programador. Releia quantas vezes forem necessárias para entender bem esse assunto.

No próximo capítulo você vai aprender sobre os três tipos de operadores que são utilizados em programação. Ok?

Mas antes vou deixar um exercício mental para você pensar. Olhe para qualquer objeto que esteja perto de você e identifique as suas características, para cada uma delas pense no tipo de dados que você utilizaria se fosse utilizar essa informação no seu software.

Por exemplo, estou olhando agora para o meu notebook e identificando algumas características nele, ele tem cor (texto), teclas (caracteres), botões de mouse para click (booleano, ou seja, pode ter dois estados "clicado" ou "não clicado"), tela (acesa ou apagada), wifi (ligado ou desligado), etc. Esse é um exercício mental que vai facilitar a sua visão sobre manipulação de dados nos seus algoritmos.

## Capítulo 4 - Operadores

Evidentemente você já resolveu alguma expressão matemática na escola. Lembra quando o professor pedia pra gente resolver aquelas expressões enormes? Cheia de números e *operadores* de somar, subtrair, multiplicar, etc. Sem falar das raízes e exponenciais. Sei que muita gente vai ter um arrepio só de lembrar disso. rs

Vamos ficar no básico ... Você sabe que na expressão  $2 + 2$  o sinal  $+$  é um operador que representa a soma. Certo? Também sabe que essa expressão resulta apenas um valor (4). Confirma? Então, você já entendeu o sentido dos *operadores*. **Relacionar valores para resultar um outro valor..**

Existem 3 tipos de *operadores*: **Aritméticos**, **Lógicos** e **Relacionais**.

Entender como funciona cada operador é MUITO importante para aprender programação. A representação simbólica de alguns operadores muda de acordo com a linguagem, mas a essência é a mesma. **Todas as linguagens de programação usam operadores.**

Vamos ver um pouquinho sobre cada tipo de operadores.

**Importante!** O conteúdo deste capítulo é um pouco denso, quero mostrar pra você como o conhecimento sobre operadores é importante. Mas você não precisa se preocupar se não entender algumas coisas que vou falar por aqui. Tome assuntos mais técnicos como curiosidade.

### Operadores Aritméticos



Esses são os mais fáceis! Aprendemos na escola fundamental. Em programação, esses operadores são tão simples quanto você aprendeu na escola. Apenas alguns que você pode não conhecer.

Além dos mais simples (soma "+", subtração "-", multiplicação "\*" e divisão "/"), dois outros operadores aritméticos

não recebem muita atenção e pode ser que você não os conheça, eles são o **div**

e o **mod**, que resultam, respectivamente, o quociente (a parte inteira do resultado da divisão) e o resto da divisão.

Peguemos por exemplo, 14 dividido por 4, vamos fazer como aprendemos na escola ...

14 dividido por 4 é igual à 3 e resta 2, certo?

Com os operadores aritméticos nós conseguimos obter o valor do quociente (4), o valor do resto (2) e o valor final da divisão (3,5).

Observe as operações abaixo:

$14 / 4 = 3,5$

$14 \text{ div } 4 = 3$

$14 \text{ mod } 4 = 2$

O operador *mod* em muitas linguagens de programação (java por exemplo) é representado pelo símbolo "%", assim:

$14 \% 4 = 2$

Uma das maiores utilizações do operador **mod** é para verificar se um número é *par* ou *ímpar*. Quando o número "**mod**" 2 resulta 0, ele é par, caso contrário, é ímpar. Veja.

$14 \text{ mod } 2 = 0$  ... **14 é par**

$15 \text{ mod } 2 = 1$  ... **15 é ímpar**

Um outro operador aritmético que existe em algumas linguagens de programação é o ^ e executa a operação de potência, mas geralmente essa operação é realizada através de uma função chamada **pow**, bem como a operação de radiciação (função **sqrt**). Veja um exemplo do operador ^:

$2 ^ 5 = 32$  (dois elevado a cinco)

Operadores aritméticos de radiciação também são fornecidos por algumas linguagens de programação, mas esses são bem mais raros. O Postgres por

exemplo oferece os símbolos `//` e `///` para operações de raiz quadrada e raiz cúbica, respectivamente.

### Precedência entre os operadores

Da mesma forma que na matemática, os operadores de multiplicação e divisão têm precedência de execução em relação aos operadores de soma e subtração. Aliás se tiver parênteses na expressão estes têm precedência ainda maior.

Os operadores de mesma prioridade são interpretados da esquerda para a direita. Para exemplificar essa questão de precedência, observe a solução da expressão abaixo:

`5 + 3 * ( 3 - 1 ) - 2 ^ 5 / 4 - 1`

`5 + 3 * 2 - 2 ^ 5 / 4 - 1`

`5 + 3 * 2 - 32 / 4 - 1`

`5 + 6 - 32 / 4 - 1`

`5 + 6 - 8 - 1`

`11 - 8 - 1`

`3 - 1`

`2`

### Operadores Lógicos

Lembra dos tipos de dados que falei no capítulo anterior? Viu que dentre os tipos de dados tinha o tipo **lógico**? Então, enquanto os operadores aritméticos trabalham com números, os operadores lógicos trabalham com dados **lógicos**, ou **booleanos**.

A ideia dos operadores continua a mesma: **Relacionar valores para resultar um outro valor..** Isso significa que os operadores lógicos relacionam valores lógicos (verdadeiro/falso, 1/0, aceso/apagado, ligado/desligado, true/false,



sim/não, etc.) e o resultado de uma expressão lógica também é outro valor lógico. Simples assim!

Vamos conhecer os operadores lógicos ... Enquanto os operadores aritméticos (que você conhece muito bem) são +, -, \*, /, mod e div, os operadores lógicos são: E, OU, NÃO, NÃO-E, NÃO-OU, OU-EXCLUSIVO e NÃO-OU-EXCLUSIVO. Não é difícil, basta você acostumar (praticando) com esses operadores. Vamos ver como funciona cada um desses.

### Operador E(AND)



O Operador "E" ou "AND" resulta em um valor VERDADEIRO se os dois valores de entrada da operação forem VERDADEIROS, caso contrário o resultado é FALSO. Abaixo a **tabela-verdade** da operação E.

VALOR 1	VALOR 2	OPERAÇÃO E
VERDADEIRO	VERDADEIRO	VERDADEIRO
VERDADEIRO	FALSO	FALSO
FALSO	VERDADEIRO	FALSO
FALSO	FALSO	FALSO

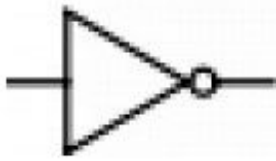
### Operador OU(OR)



O Operador "OU" ou "OR" resulta em um valor VERDADEIRO se ao menos UM dos dois valores de entrada da operação for VERDADEIRO, caso contrário o resultado é FALSO. Abaixo a **tabela-verdade** da operação OU.

VALOR 1	VALOR 2	OPERAÇÃO OU
VERDADEIRO	VERDADEIRO	VERDADEIRO
VERDADEIRO	FALSO	VERDADEIRO
FALSO	VERDADEIRO	VERDADEIRO
FALSO	FALSO	FALSO

### Operador NÃO(NOT)



O Operador "NÃO" ou "NOT" é o único operador que recebe como entrada apenas um valor, e sua função é simplesmente inverter os valores. Ou seja, se o valor de entrada for VERDADEIRO, o resultado será FALSO e se o valor de entrada for FALSO, o resultado será VERDADEIRO. Abaixo a tabela-verdade da operação NÃO.

VALOR DE ENTRADA	OPERAÇÃO OU
VERDADEIRO	FALSO
FALSO	VERDADEIRO

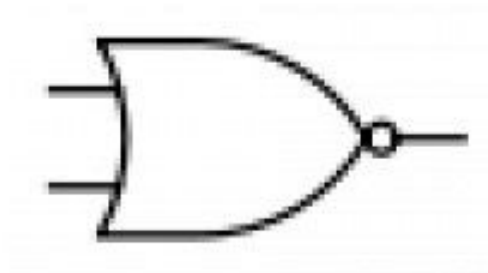
### Operador NÃO-E(NAND)



O Operador "NÃO-E" ou "NAND" é o contrário do operador E (AND), ou seja, resulta em VERDADEIRO, se ao menos um dos dois valores for FALSO, na verdade este é o operador E (AND) seguido do operador NÃO (NOT). Abaixo a **tabela-verdade** da operação NÃO-E.

VALOR 1	VALOR 2	OPERAÇÃO NÃO-E
VERDADEIRO	VERDADEIRO	FALSO
VERDADEIRO	FALSO	VERDADEIRO
FALSO	VERDADEIRO	VERDADEIRO
FALSO	FALSO	VERDADEIRO

### Operador NÃO-OU(NOR)



O Operador "NÃO-OU" ou "NOR" é o contrário do operador OU (OR), ou seja, resulta em VERDADEIRO, se os dois valores forem FALSO, na verdade este é o operador OU (OR) seguido do operador NÃO (NOT). Abaixo a **tabela-verdade** da operação NÃO-OU.

VALOR 1	VALOR 2	OPERAÇÃO NÃO-OU
VERDADEIRO	VERDADEIRO	FALSO
VERDADEIRO	FALSO	FALSO
FALSO	VERDADEIRO	FALSO
FALSO	FALSO	VERDADEIRO

### Operador OU-EXCLUSIVO(XOR)



O Operador "OU-EXCLUSIVO" ou "XOR" é uma variação interessante do operador OU (OR), ele resulta em VERDADEIRO se apenas um dos valores de entrada for VERDADEIRO, ou seja, apenas se os valores de entrada forem DIFERENTES. Abaixo a tabela-verdade da operação OU-EXCLUSIVO.

VALOR 1	VALOR 2	OPERAÇÃO OU-EXCLUSIVO
VERDADEIRO	VERDADEIRO	FALSO
VERDADEIRO	FALSO	VERDADEIRO
FALSO	VERDADEIRO	VERDADEIRO
FALSO**	FALSO	FALSO

### Operador NÃO-OU-EXCLUSIVO(XNOR)



O Operador "NÃO-OU-EXCLUSIVO" ou "XNOR" é o contrário do operador OU-EXCLUSIVO (XOR), ou seja, resulta VERDADEIRO se os valores de entrada forem IGUAIS. Observe a tabela abaixo:

VALOR 1	VALOR 2	OPERAÇÃO NÃO-OU-EXCLUSIVO
VERDADEIRO	VERDADEIRO	VERDADEIRO
VERDADEIRO	FALSO	FALSO
FALSO	VERDADEIRO	FALSO
FALSO	FALSO	VERDADEIRO

### Operadores lógicos em programação

Cada linguagem de programação tem uma forma de representar os operadores lógicos. A simbologia mais encontrada são:

- **AND, OR e NOT** em linguagens como: Pascal, Visual Basic e SQL.
- **&&, || e !** em linguagens como: Java e C#

No nosso caso, criando algoritmos em português, os operadores lógicos são E, OU, etc. Por exemplo.

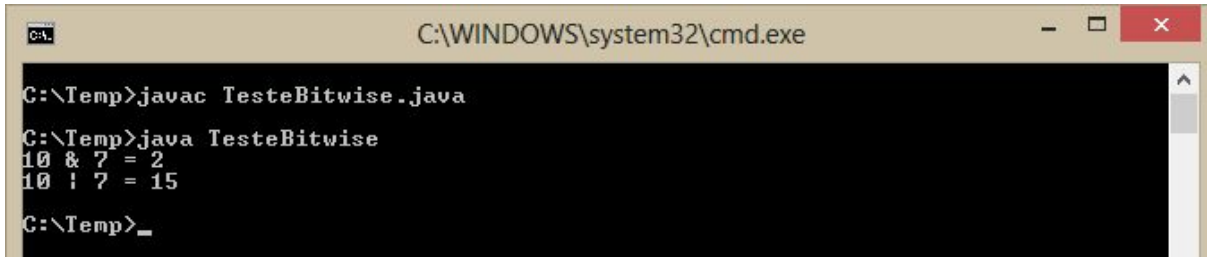
VERDADEIRO **E** FALSO = FALSO

Algumas linguagens oferecem operadores lógicos para o nível de bit (também chamado de operadores bitwise). Ou seja, podemos fazer operações lógicas com os bits de dois números. Em java, por exemplo esses operadores são & e |. Veja o código abaixo escrito em java.

```
public class TesteBitwise {  
    public static void main (String []a){  
        System.out.println("10 & 7 = " + (10 & 7));  
        System.out.println("10 | 7 = " + (10 | 7));  
    }  
}
```

```
}  
}
```

Abaixo o resultado deste programa:



```
C:\WINDOWS\system32\cmd.exe  
C:\Temp>javac TesteBitwise.java  
C:\Temp>java TesteBitwise  
10 & 7 = 2  
10 | 7 = 15  
C:\Temp>_
```

Essas operações lógicas são realizadas com os bits dos números de entrada. Assim:

Convertemos o número 10 e o número 7 para binário:

10 = 1010 em binário

7 = 0111 em binário

depois realizamos as operações lógicas com cada bit dos dois números.

Da direita para a esquerda aplicamos as operações lógicas para cada bit.

0 E 1 = 0

1 E 1 = 1

0 E 1 = 0

1 E 0 = 0

Logo,

$10 \& 7 = 0010 = 2$  em números decimais.

O mesmo para o operador OU:

$0 \text{ OU } 1 = 1$

$1 \text{ OU } 1 = 1$

$0 \text{ OU } 1 = 1$

$1 \text{ OU } 0 = 1$

Logo,

$10 | 7 = 1111 = 15$

Não é o foco aqui deste e-book, essa parte das operações em nível de bit foi só para curiosidade. Você não precisa saber fazer conversões de bases numéricas para aprender lógica de programação, mas se quiser aprender um pouco mais sobre isso, leia este post que escrevi no blog { **Dicas de Programação** }:

[As 10 conversões numéricas mais utilizadas na computação](#)

## Operadores Relacionais

Operadores relacionais são utilizados para comparar valores (de qualquer tipo), o resultado de uma expressão relacional é um valor booleano (VERDADEIRO ou FALSO). Os operadores relacionais são: **igual, diferente, maior, menor, maior ou igual, menor ou igual**.

Não é necessário explicar cada um, pois eles são auto-explicativos. Mas para quem é iniciante em desenvolvimento de softwares algumas informações podem ser importantes, principalmente pelo fato de haver diferença entre linguagens de programação.

Os operadores relacionais são diferente dependendo da linguagem de programação, mas conhecendo os símbolos mais comuns fica mais fácil



aprender. No [VisuAlg](#), os símbolos dos operadores relacionais são: =, <>, >, <, >=, <=. Vamos testar esses operadores no VisuAlg com o algoritmo abaixo que compara dois números com cada um dos operadores e mostra o resultado na tela:

```
algoritmo "TesteOperadoresRelacionais"
var
    numero1 : INTEIRO
    numero2 : INTEIRO
    resultado : LOGICO
inicio

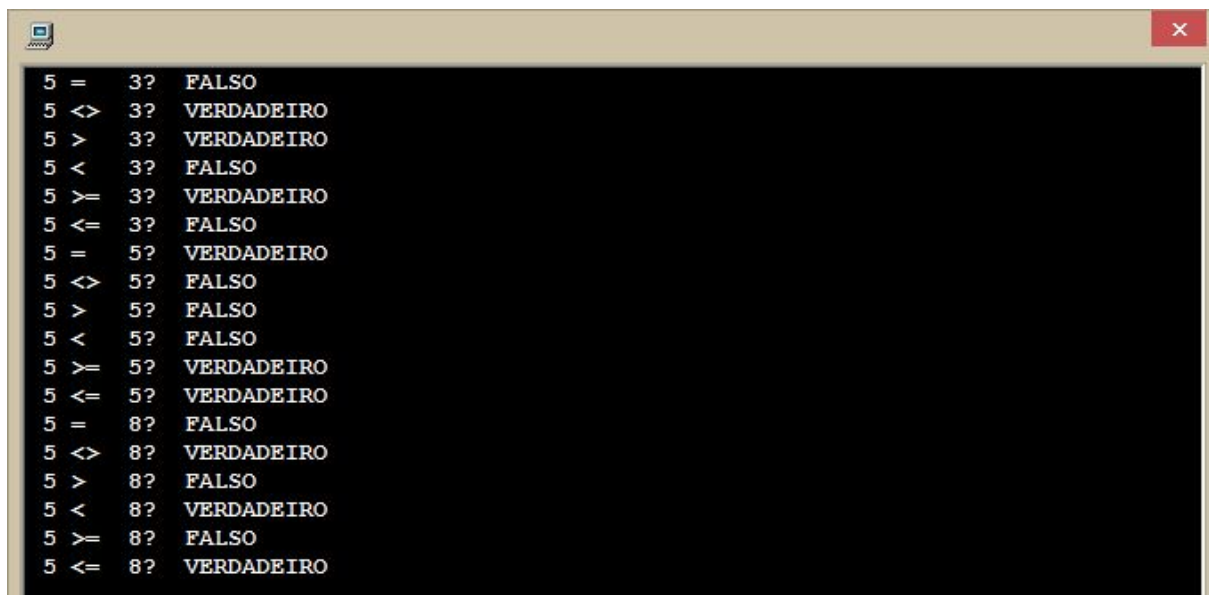
    numero1 := 5
    numero2 := 3
    resultado := numero1 = numero2
    ESCREVAL (numero1, " = ", numero2, "? ", resultado)
    resultado := numero1 <> numero2
    ESCREVAL (numero1, " <> ", numero2, "? ", resultado)
    resultado := numero1 > numero2
    ESCREVAL (numero1, " > ", numero2, "? ", resultado)
    resultado := numero1 < numero2
    ESCREVAL (numero1, " < ", numero2, "? ", resultado)
    resultado := numero1 >= numero2
    ESCREVAL (numero1, " >= ", numero2, "? ", resultado)
    resultado := numero1 <= numero2
    ESCREVAL (numero1, " <= ", numero2, "? ", resultado)

    numero1 := 5
    numero2 := 5
    resultado := numero1 = numero2
    ESCREVAL (numero1, " = ", numero2, "? ", resultado)
    resultado := numero1 <> numero2
    ESCREVAL (numero1, " <> ", numero2, "? ", resultado)
    resultado := numero1 > numero2
    ESCREVAL (numero1, " > ", numero2, "? ", resultado)
    resultado := numero1 < numero2
    ESCREVAL (numero1, " < ", numero2, "? ", resultado)
    resultado := numero1 >= numero2
    ESCREVAL (numero1, " >= ", numero2, "? ", resultado)
    resultado := numero1 <= numero2
    ESCREVAL (numero1, " <= ", numero2, "? ", resultado)
```

```
numero1 := 5
numero2 := 8
resultado := numero1 = numero2
ESCREVAL (numero1, " = ", numero2, "? ", resultado)
resultado := numero1 <> numero2
ESCREVAL (numero1, " <> ", numero2, "? ", resultado)
resultado := numero1 > numero2
ESCREVAL (numero1, " > ", numero2, "? ", resultado)
resultado := numero1 < numero2
ESCREVAL (numero1, " < ", numero2, "? ", resultado)
resultado := numero1 >= numero2
ESCREVAL (numero1, " >= ", numero2, "? ", resultado)
resultado := numero1 <= numero2
ESCREVAL (numero1, " <= ", numero2, "? ", resultado)
```

fimalgoritmo

A intenção deste algoritmo é mostrar o funcionamento dos operadores relacionais com 3 possibilidades de valores: um número menor que o outro, dois números iguais e um número maior que outro. Abaixo o resultado da execução:



```
5 = 3? FALSO
5 <> 3? VERDADEIRO
5 > 3? VERDADEIRO
5 < 3? FALSO
5 >= 3? VERDADEIRO
5 <= 3? FALSO
5 = 5? VERDADEIRO
5 <> 5? FALSO
5 > 5? FALSO
5 < 5? FALSO
5 >= 5? VERDADEIRO
5 <= 5? VERDADEIRO
5 = 8? FALSO
5 <> 8? VERDADEIRO
5 > 8? FALSO
5 < 8? VERDADEIRO
5 >= 8? FALSO
5 <= 8? VERDADEIRO
```

## Operadores lógicos em programação

Em todas as linguagens de programação existem símbolos para executarmos essas operações. As operações maior, menor, maior ou igual e menor ou igual na maioria das linguagens de programação são os mesmos símbolos (até hoje não encontrei uma linguagem que tenha símbolo diferente para estes operadores): > (maior), < (menor), >= (maior ou igual) e <= (menor ou igual).

Mas os vilões dos iniciantes são os símbolos para testar igualdade e diferença. Em cada linguagem é de um jeito! Em java, C, C#, javascript,... por exemplo, os símbolos de igual e diferente são: == e !=. Já em Pascal, SQL, Visual Basic, ... os símbolos de igual e diferente são: = e <>. Então fique esperto quando for aprender alguma dessas linguagens!

Em java não é possível testar Strings (textos) com o operador de igualdade (==), pois String é uma classe e não um tipo primitivo, e para testar a igualdade entre objetos deve-se utilizar o método *equals()*. Assim:

```
nome.equals("João").
```

Em algumas linguagens de programação (Python por exemplo) é possível utilizar os operadores maior e menor para verificar a precedência alfabética de um texto em relação a outro. Por exemplo: "Pedro" < "Paulo" resulta em FALSO, pois o texto "Pedro" alfabeticamente aparece depois do texto "Paulo".

Sei que neste capítulo teve bastante informação técnica e talvez você não tenha entendido algumas coisas. Não se preocupe. Se você entendeu o que são os operadores e como utilizamos eles na programação, está ótimo.

Não precisa aprofundar nos assuntos que tratamos aqui (bitwise, por exemplo), só queria te mostrar que os operadores são muito utilizados e não conhecer pelo menos o básico sobre os operadores pode comprometer o seu aprendizado de programação.

Importante lembrar que do mesmo jeito que aprendemos os operadores aritméticos nas escola, para aprender os operadores relacionais e lógicos (menos comuns) é necessário bastante prática!

Por isso, após o próximo capítulo onde você vai aprender a estrutura SE-ENTÃO-SENÃO, você poderá praticar bastante o uso desses operadores com exercícios.

## Capítulo 5 - Tomando decisões!

Agora vamos colocar a mão na massa e aprender a forma mais básica de controlar o fluxo de um algoritmo.

Vamos fazer os nossos algoritmos tomarem decisões!

Para isso existem as estruturas de decisão, e a mais utilizada é a estrutura SE-ENTÃO-SENÃO (Em inglês IF-THEN-ELSE).

### Estrutura de decisão SE-ENTÃO-SENÃO

O funcionamento é simples: com base no resultado de uma expressão lógica (lembra do último capítulo quando falamos dos operadores lógicos?), o fluxo do algoritmo segue para um bloco de instruções ou não. Observe o esquema da estrutura SE-ENTÃO-SENÃO:

```
SE <expressão lógica>  
  ENTÃO  
    <instruções a serem executadas caso a expressão booleana resulte em VERDADEIRO>  
  SENÃO  
    <instruções a serem executadas caso a expressão booleana resulte em FALSO>  
FIM-SE
```

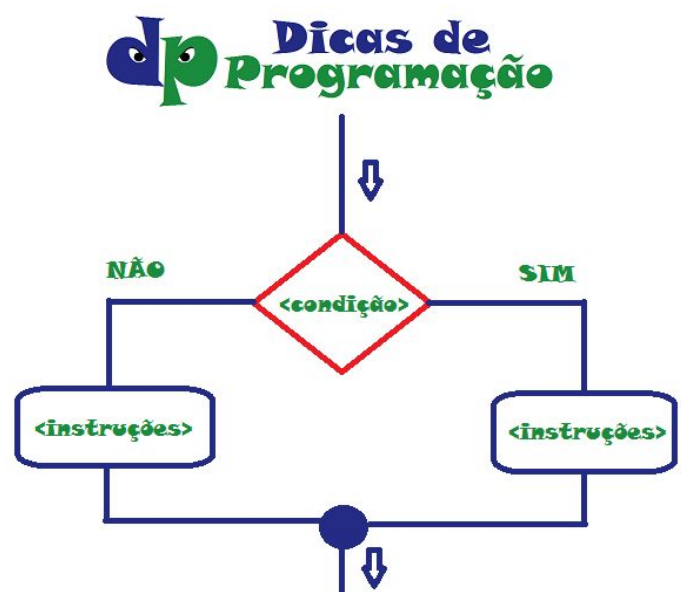
O bloco de código SENÃO é opcional. É comum encontrar instruções de decisão apenas com SE-ENTÃO sem o bloco SENÃO. Veja um esquema gráfico desta estrutura de decisão:

Simples assim. Essa estrutura não tem segredos. Agora é hora de praticar! Vamos lá?

### SE-ENTÃO-SENÃO na prática!

Vejamos um exemplo de utilização desta estrutura com um algoritmo, você pode usar o [VisuAlg](https://visuAlg.com.br) para testar esse algoritmo e ver o resultado.

Neste algoritmo, vamos simular um caixa eletrônico quando vamos sacar dinheiro. O caixa eletrônico verifica se



o valor que desejamos sacar é menor que o saldo disponível. Assumiremos que há R\$ 1000 de saldo disponível para o saque. Se o valor que o usuário quer sacar é menor ou igual ao saldo disponível, então o algoritmo permite o saque, caso contrário, não.

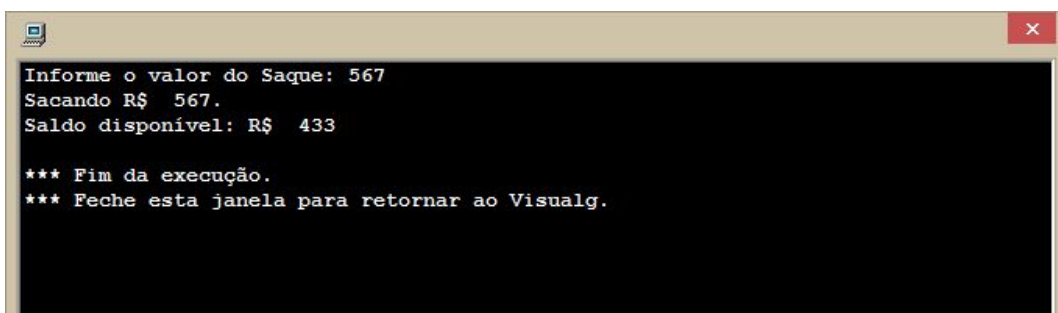
```
algoritmo "SacarDinheiro"
var
    SaldoDisponivel : REAL
    ValorDoSaque : REAL
inicio

    SaldoDisponivel := 1000 //Assumimos que há 1000 reais de saldo na
conta disponível para saque
    ESCREVA ("Informe o valor do Saque: ")
    LEIA (ValorDoSaque)
    SE ValorDoSaque <= SaldoDisponivel ENTAO
        SaldoDisponivel := SaldoDisponivel - ValorDoSaque
        ESCREVAL ("Sacando R$ ", ValorDoSaque, ".")
    SENAO
        ESCREVAL ("O valor solicitado é maior que o valor disponível para
saque!")
    FIMSE

    ESCREVAL ("Saldo disponível: R$ ", SaldoDisponivel)

finalgoritmo
```

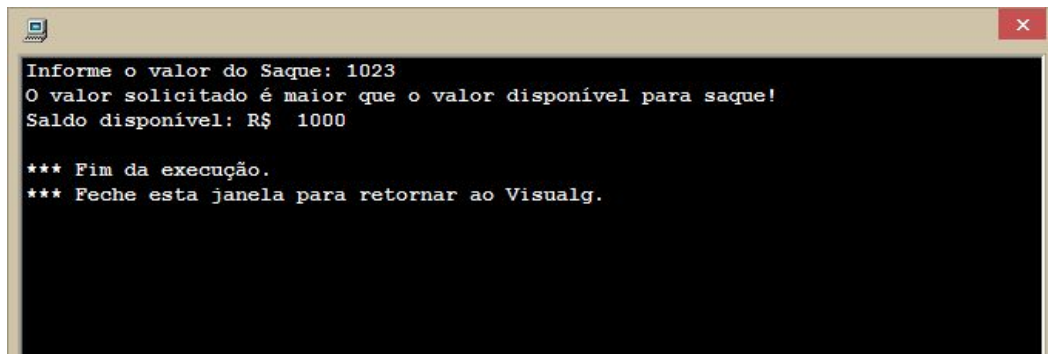
Abaixo a execução do algoritmo acima quando informamos valores menores que 1000:



```
Informe o valor do Saque: 567
Sacando R$ 567.
Saldo disponível: R$ 433

*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

Agora a execução do mesmo algoritmo, porém inserindo um valor maior que 1000 para saque:



```
Informe o valor do Saque: 1023
O valor solicitado é maior que o valor disponível para saque!
Saldo disponível: R$ 1000

*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

Perceba que o fluxo do algoritmo tomou rumos diferentes.

Essa é a estrutura de controle de fluxo mais utilizada na criação de programas de computador. Pratique-a criando algoritmos que tomam decisão.

### Hora de praticar!

Lembra do exercício que você fez no capítulo 2? Aquele que calcula a média de um aluno. Vamos incrementar ele e informar se ele foi aprovado ou reprovado. Então o algoritmo deve ser assim: O usuário digita as 4 notas (de 0 a 10) bimestrais do aluno e o algoritmo deve calcular a média. Depois o algoritmo deve verificar se a média é maior ou igual a 6. **Caso afirmativo**, exibe na tela uma mensagem informando que o aluno foi aprovado, **caso contrário**, uma mensagem informando que ele foi reprovado. No próximo capítulo eu vou mostrar o meu algoritmo para solucionar este exercício. Mas é muito importante que você tente fazer esse algoritmo sozinho antes de ver a resposta. Ok? Além disso você vai aprender como fazemos para nosso algoritmo tomar decisão quando tem MUITAS opções.

## Capítulo 6: Tomando decisões entre muitas opções

Olá querido leitor, conseguiu resolver o exercício do último capítulo? No final deste capítulo você poderá ver a minha solução para você conferir com o seu algoritmo.

No último capítulo nós falamos da estrutura **SE-ENTÃO-SENÃO**, que é usada para fazer os nossos programas tomarem decisões por si só.

Dei o exemplo do caixa eletrônico, em que o programa deveria verificar se o valor que desejamos sacar é menor que o saldo disponível.

Neste capítulo vamos ver qual estrutura de controle de fluxo devemos utilizar quando temos muitas opções para tomar decisão.

Antes de aprender a estrutura ESCOLHA-CASO, vamos ver uma coisa que a princípio não tem nada a ver com o nosso assunto, mas vai te ajudar a entender como funciona esta estrutura.

### Equipamentos de rede de computadores

Talvez você já saiba mais ou menos algumas coisas sobre rede de computadores. Existem várias topologias de redes: estrela, barramento, anel, etc...

Mas quero chamar a sua atenção para dois equipamentos utilizados nas redes de computadores. Um é o **HUB** e o outro é o **SWITCH**.



Esses dois equipamentos são muito parecidos, algumas pessoas até pensam que são a mesma coisa. Mas há uma pequena diferença entre eles.



A tarefa é a mesma, transferir dados entre as portas, a diferença é a forma com que a tarefa é realizada por cada equipamento.

Basicamente o **HUB é burro** e o **SWITCH é inteligente**. Como assim?

Simples, quando o HUB recebe dados por uma porta, ele reenvia esses dados para TODAS as portas.

Por exemplo, se o computador ligado na porta 1 enviou um pacote de dados para o computador ligado na porta 5, o HUB enviará os dados para todas as portas, 1, 2, 3, 4, 5, 6... O computador de destino que vai descobrir se o pacote de dados é pra ele ou não, caso não o seja ele vai ignorar o pacote de dados.

Isso significa que os HUBs deixam a rede lenta, pois haverá muito congestionamento de dados na rede e processamento desnecessário pelos computadores. Além disso, apenas um pacote estará trafegando na rede por vez.

Ou seja, o **HUB é burro**!

Já o SWITCH é mais inteligente. Quando o SWITCH recebe um pacote de dados, ele identifica a porta correta para encaminhar aquele pacote de dados.

Por exemplo, se o computador ligado na porta 1 enviou um pacote de dados para o computador ligado na porta 5, o SWITCH enviará os dados apenas para a porta 5.

Dessa forma há menos congestionamento na rede e é possível trafegar vários pacotes na rede paralelamente.

### Lembra que inglês é importante?

Talvez você esteja se perguntando o que tem a ver o HUB e o SWITCH com o assunto deste capítulo. Tudo!

A estrutura ESCOLHA-CASO funciona da mesma forma que o SWITCH das redes de computadores só que ao invés de enviar um pacote de dados para uma determinada porta, vamos enviar o fluxo do algoritmo para um determinado ponto do código. A ideia é a mesma!

A propósito, como eu disse no primeiro capítulo, inglês é essencial para trabalhar com programação, EMBORA NÃO SEJA IMPEDITIVO. E quando você estiver programando em inglês verá que ESCOLHA-CASO é conhecido como SWITCH-CASE.

### A estrutura ESCOLHA-CASO

Lembra do SE-ENTÃO-SENÃO do capítulo passado? Imagine que você tem um menu de opções e o usuário deve escolher uma opção, dentre várias. Como você identificaria qual opção o usuário digitou? Talvez você faria algo assim ...

```
SE opção = 1 ENTÃO
    "instruções a serem executadas caso opção = 1"
SENÃO
    SE opção = 2 ENTÃO
        "instruções a serem executadas caso opção = 2"
    SENÃO
        SE opção = 3 ENTÃO
            "instruções a serem executadas caso opção = 3"
        SENÃO
            ...
        FIM-SE
    FIM-SE
FIM-SE
```

Ou seja, vários SE-ENTÃO-SENÃO aninhados, um no SENÃO do outro...

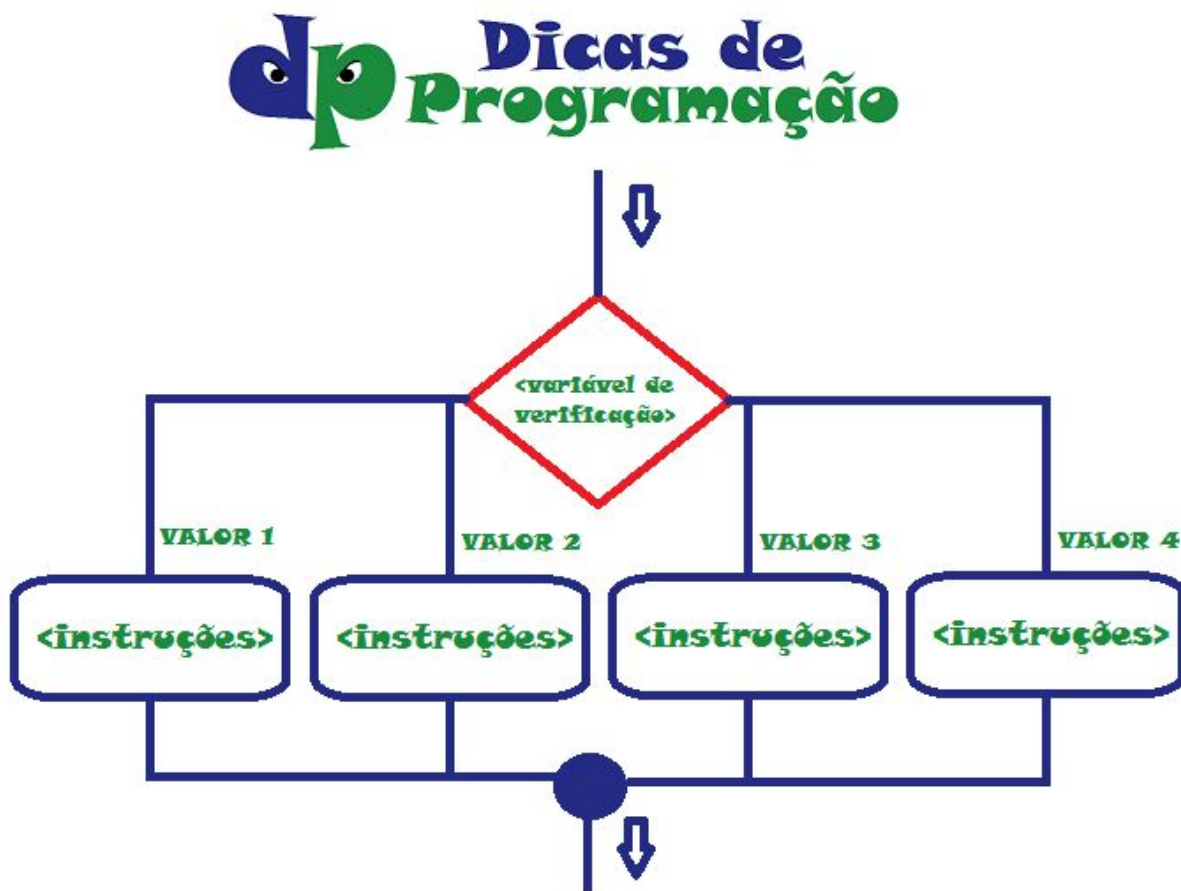
A proposta do ESCOLHA-CASO é ser uma solução mais elegante para este caso. Levando o fluxo do programa direto ao bloco de código correto (igual o switch), dependendo do valor de uma variável de verificação.

Essa é a estrutura ESCOLHA-CASO.

```
ESCOLHA <variável de verificação>
    CASO <valor1> FAÇA
        "instruções a serem executadas caso <variável de
verificação> = <valor1>"
    CASO <valor2> FAÇA
        "instruções a serem executadas caso <variável de
verificação> = <valor2>"
    CASO <valor3> FAÇA
        "instruções a serem executadas caso <variável de
verificação> = <valor3>"
    ...
```

FIM-ESCOLHA

O esquema visual do fluxograma desta estrutura é como a figura abaixo:



### ESCOLHA-CASO na prática!

Nada melhor para aprender programação do que praticar. Bastante! Então vamos ver um exemplo prático da utilização do ESCOLHA-CASO em comparação ao SE-ENTÃO-SENÃO.

(Novamente vamos usar o Visualg para criar os nossos algoritmos, você pode fazer com lápis e papel, mas caso queira baixar o Visualg, [clique aqui para fazer o download](#))

Imagine a seguinte situação: Você deseja criar um algoritmo para uma calculadora, o usuário digita o primeiro número, a operação que deseja executar e o segundo número. Dependendo do que o usuário informar como operador, o

algoritmo executará um cálculo diferente (soma, subtração, multiplicação ou divisão).

Vejamos como seria este algoritmo utilizando a estrutura SE-ENTÃO-SENÃO:

```
algoritmo "CalculadoraBasicaComSE"
var
    numero1 : REAL
    numero2 : REAL
    operacao : CARACTERE
    resultado : REAL
inicio

    ESCREVA ("Digite o primeiro número: ")
    LEIA (numero1)
    ESCREVA ("Digite a operação: ")
    LEIA (operacao)
    ESCREVA ("Digite o segundo número: ")
    LEIA (numero2)

    SE operacao = "+" ENTÃO
        resultado := numero1 + numero2
    SENÃO
        SE operacao = "-" ENTÃO
            resultado := numero1 - numero2
        SENÃO
            SE operacao = "*" ENTÃO
                resultado := numero1 * numero2
            SENÃO
                SE operacao = "/" ENTÃO
                    resultado := numero1 / numero2
                FIMSE
            FIMSE
        FIMSE
    FIMSE

    ESCREVA ("Resultado: ", resultado)
```

fimalgoritmo

Veja como os SEs aninhados (dentro dos SENÃOs) deixam o código mais complexo. Dá pra entender a lógica, mas não é muito elegante. Agora vamos ver como ficaria a mesma lógica com a estrutura ESCOLHA-CASO.

```
algoritmo "CalculadoraBasicaComESCOLHA_CASO"
var
    numero1 : REAL
    numero2 : REAL
    operacao : CARACTERE
    resultado : REAL
inicio

    ESCREVA ("Digite o primeiro número: ")
    LEIA (numero1)
    ESCREVA ("Digite a operação: ")
    LEIA (operacao)
    ESCREVA ("Digite o segundo número: ")
    LEIA (numero2)

    ESCOLHA operacao
        CASO "+"
            resultado := numero1 + numero2
        CASO "-"
            resultado := numero1 - numero2
        CASO "*"
            resultado := numero1 * numero2
        CASO "/"
            resultado := numero1 / numero2
    FIMESCOLHA

    ESCREVA ("Resultado: ", resultado)

fimalgoritmo
```

Bem mais bonito! Né? Agora a lógica tá mais visível e elegante. O resultado dos dois algoritmos é o mesmo. Mas o código com o ESCOLHA-CASO é mais fácil de entender.

### CASO NÃO TRATADO NA ESTRUTURA (OUTROCASO)

Além das opções tratadas na estrutura, é possível identificar quando o valor da variável não é equivalente a nenhum valor informado como opção nos CASOs, ou seja, é um "OUTROCASO".

No algoritmo que fizemos anteriormente, imagine se o usuário digitasse um valor diferente de "+", "-", "\*" e "/". Caso quiséssemos apresentar uma mensagem para o usuário informando que ele digitou uma opção inválida, utilizaríamos esse recurso da estrutura ESCOLHA-CASO. Veja:

```
ESCOLHA operacao
  CASO "+"
    resultado := numero1 + numero2
  CASO "-"
    resultado := numero1 - numero2
  CASO "*"
    resultado := numero1 * numero2
  CASO "/"
    resultado := numero1 / numero2
  OUTROCASO
    ESCREVA("A operação digitada é inválida!")
FIMESCOLHA
```

Como você pôde observar, em termos de organização de código a estrutura ESCOLHA-CASO é uma opção muito elegante quando se tem muitos SE-ENTÃO-SENÃO para verificar a mesma variável. Facilita a leitura do algoritmo e a manutenção do código.

### Exercício

Aprender programação é como aprender matemática, tem que praticar muito fazendo exercícios. Portanto vou deixar mais um exercício para você resolver sozinho, com o assunto que vimos neste capítulo.

\* Crie um algoritmo em que o usuário digita uma letra qualquer e o programa verifica qual a ordem dessa letra no alfabeto, por exemplo: se o usuário digitar a letra 'G' o programa deve imprimir na tela, "A letra G está na posição 7 do

alfabeto". Implemente com a estrutura ESCOLHA-CASO e depois com a estrutura SE-ENTÃO-SENÃO para perceber a diferença gritante no código.

É muito importante que você tente fazer os algoritmos sozinho antes de ver a resposta. Ok?



## Capítulo 7 - Loops básicos!

Até aqui aprendemos sobre escrever dados na tela e ler informações que o usuário digita, aprendemos o que são variáveis e constantes, aprendemos um pouco mais sobre operadores (aritméticos, lógicos e relacionais) e também aprendemos a controlar o fluxo de um algoritmo. Através de estruturas de decisões e de seleção decidimos para onde o fluxo do nosso algoritmo deve seguir.

Com o que aprendemos até agora já dá pra fazer muita coisa com programação!

Mas agora vamos aprender um recurso MUITO usado na programação: Os LOOPS. Entender como funcionam os LOOPS na programação fará você mudar a forma de pensar em algoritmos.

### O que é LOOP?

Lembra quando você aprendeu a fazer multiplicação?

O(A) professor(a) deve ter te ensinado a fazer várias somas. Certo?

Por exemplo ...

$$4 * 5 = 4 + 4 + 4 + 4 + 4$$

Nosso(a) professor(a), nos ensinou a fazer um loop!

Em programação, LOOP é uma instrução para o programa repetir tarefas.

No algoritmo da multiplicação, nós somamos o primeiro valor X vezes, sendo X o segundo valor.

Os loops são muito utilizados no mundo da programação. Eles vêm em 3 sabores: ENQUANTO-FAÇA, REPITA-ATÉ e PARA-FAÇA.

Neste capítulo vamos estudar os dois primeiros: ENQUANTO-FAÇA e REPITA-ATÉ.

### Estrutura de repetição ENQUANTO-FAÇA

O funcionamento da estrutura de repetição ENQUANTO-FAÇA (em inglês WHILE-DO) é tão simples quanto o SE-ENTÃO-SENÃO. A diferença é que os passos dentro deste bloco são repetidos enquanto a expressão booleana resultar VERDADEIRO.

Obs: Lembrando os tipos de dados do capítulo 3, o tipo de dados booleano só pode assumir dois valores: VERDADEIRO ou FALSO.

Voltando ao ENQUANTO ... Vejamos como ficaria o pseudo-código desta estrutura:

ENQUANTO <expressão booleana> FAÇA

    <instruções a serem executadas enquanto a expressão booleana resultar em VERDADEIRO>

FIM-ENQUANTO

Também chamamos esta estrutura de repetição de *loop pré-testado*, pois a expressão booleana é verificada antes da primeira execução. Se inicialmente ela já resultar em FALSO, as instruções que estão dentro do bloco não são executadas nenhuma vez.

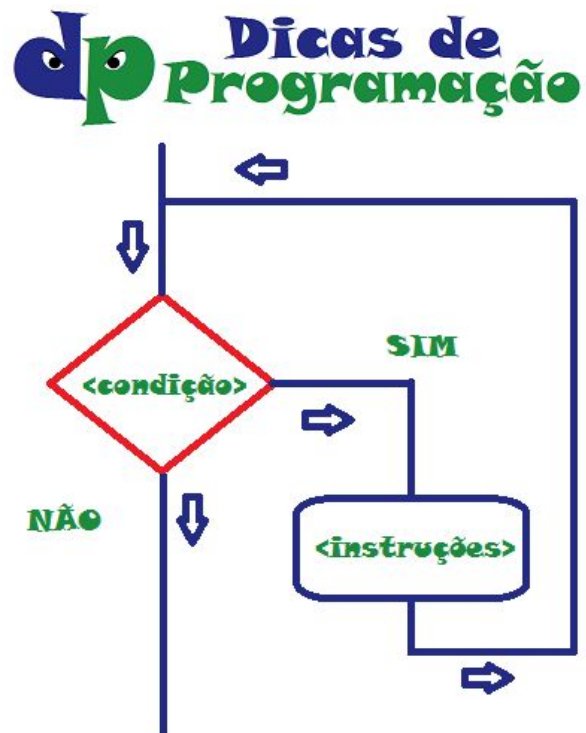
Este é o fluxograma desta estrutura de repetição. Repare que testamos a condição antes de entrar no LOOP:

### Hora de praticar!

Para aprender programação, nada melhor que praticar! Vamos ver um exemplo de LOOP com a estrutura ENQUANTO-FAÇA, utilizando a ferramenta VisuAlg.

Vamos fazer um algoritmo para somar valores até o usuário digitar o valor 0. Ou seja, vamos somar todos os valores que o usuário digitar, porém quando ele digitar 0 o "loop" acaba, a cada iteração do loop vamos apresentar o resultado atual da soma.

```
algoritmo "SomaEnquantoValorDiferenteDe0"  
var
```



```
valorDigitado : REAL
soma : REAL
inicio

    soma := 0
    ESCREVA ("Digite um valor para a soma: ")
    LEIA (valorDigitado)

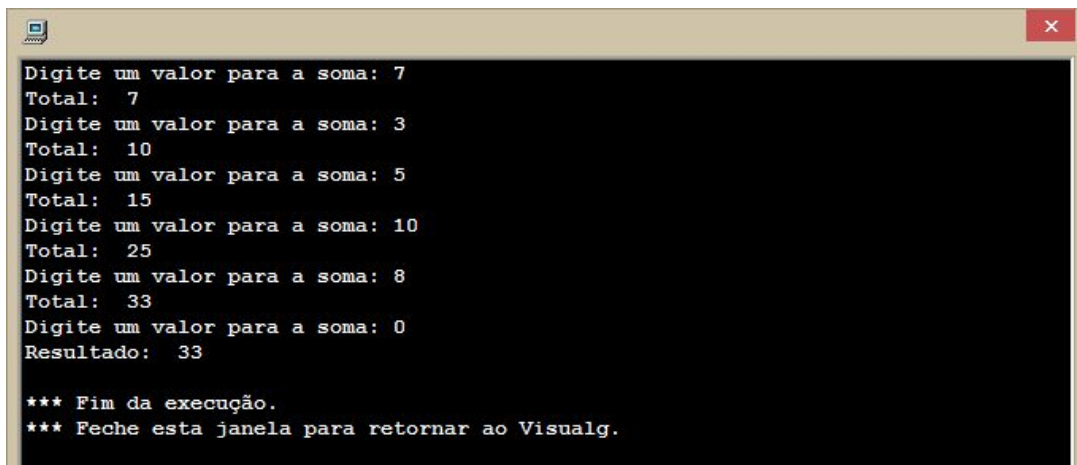
    ENQUANTO valorDigitado <> 0 FACA
        soma := soma + valorDigitado
        ESCREVAL ("Total: ", soma)
        ESCREVA ("Digite um valor para a soma: ")
        LEIA (valorDigitado)
    FIMENQUANTO

    ESCREVAL ("Resultado: ", soma)

finalgoritmo
```

*Obs. A função ESCREVAL quebra a linha (como um ENTER) no final.*

O resultado deste algoritmo é algo assim:



```
Digite um valor para a soma: 7
Total: 7
Digite um valor para a soma: 3
Total: 10
Digite um valor para a soma: 5
Total: 15
Digite um valor para a soma: 10
Total: 25
Digite um valor para a soma: 8
Total: 33
Digite um valor para a soma: 0
Resultado: 33

*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

## Estrutura de repetição REPITA-ATÉ

Acho que você já deve imaginar como é esta estrutura né? Não!? Fácil!

Lembra que eu disse que a estrutura ENQUANTO-FAÇA é conhecida como *loop pré-testado*. Então, a estrutura REPITA-ATÉ (REPEAT-UNTIL em inglês) é o contrário. Ela é um LOOP *pós-testado*. Isso significa que a verificação para repetir o LOOP é testada no final do bloco.

Este é o pseudo-código do REPITA-ATÉ:

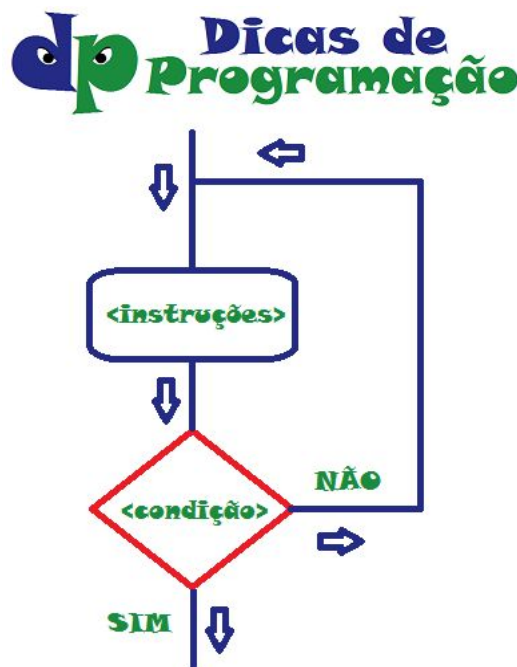
REPITA

<instruções a serem executadas repetidamente até a expressão booleana retornar VERDADEIRO>

ATÉ <expressão booleana>

Uma coisa muito importante a se notar é que além de ser pós-testada, esta estrutura testa o contrário do ENQUANTO. Ou seja, na estrutura REPITA-ATÉ, as instruções do bloco são executadas repetidamente enquanto a expressão booleana resultar FALSO. A partir do momento que a expressão booleana resultar VERDADEIRO, o fluxo do algoritmo sairá do LOOP.

Veja o funcionamento no fluxograma:



Não sei se você também percebeu, enquanto na estrutura ENQUANTO-FAÇA o bloco do LOOP pode não ser executado nenhuma vez, na estrutura REPITA-ATÉ o bloco é executado pelo menos uma vez.

### Hora de praticar!

Que tal fazer o mesmo exercício que fizemos acima com a estrutura ENQUANTO-FAÇA, mas desta vez utilizando a estrutura REPITA-ATÉ? Vamos ver como ficaria?

```
algoritmo "SomaAteValorIgualA0"
var
    valorDigitado : REAL
    soma : REAL
inicio
    soma := 0

    REPITA
        ESCREVA ("Digite um valor para a soma: ")
        LEIA (valorDigitado)
        soma := soma + valorDigitado
        ESCREVAL ("Total: ", soma)
    ATE valorDigitado = 0

finalgoritmo
```

### Algumas diferenças ...

Se você prestar atenção, vai perceber que na estrutura ENQUANTO-FAÇA tivemos que repetir uma parte do código antes do LOOP e dentro do LOOP. Repetimos a seguinte parte:

```
    ESCREVA ("Digite um valor para a soma: ")
    LEIA (valorDigitado)
```

Isso aconteceu porque a estrutura ENQUANTO-FAÇA é pré-testada. Não daria pra testar se o usuário digitou o valor 0 se ele ainda não tivesse digitado valor nenhum.

Na estrutura REPITA-ATÉ não precisamos escrever essas duas linhas duas vezes, pois ela é pós-testada.

Ah! Outra coisa que também não pode ser deixada de lado é que agora o teste de verificação do LOOP mudou de  $(\text{valorDigitado} < > 0)$  na estrutura ENQUANTO, para  $(\text{valorDigitado} = 0)$  na estrutura REPITA-ATÉ.

Você saberia explicar por quê? Pense um pouco e responda por si mesmo. O resultado deste algoritmo é o mesmo do anterior.

### Conclusão

Percebemos que é possível utilizar qualquer uma das duas estruturas para implementar LOOPS, porém cada uma é mais apropriada dependendo do problema. Neste problema em particular, a estrutura REPITA-ATÉ se mostrou mais apropriada. Uma vez que nesta estrutura não é necessário repetir um pedaço do código.

A decisão de qual estrutura utilizar entre as duas, sempre será tomada observando a diferença entre PRÉ-TESTADA e PÓS-TESTADA. Fora isso é gosto pessoal (ou requisito do chefe para padronizar o código).

Aprenda muito bem os LOOPS! As estruturas de repetição são muito utilizadas em desenvolvimento de softwares. Entender como elas funcionam é muito importante para resolver problemas que precisam executar tarefas repetidas vezes. Acredite, existem muitos!

Para praticar a utilização da estrutura ENQUANTO, um exercício!

Lembra da multiplicação do começo do capítulo? Quero que você faça um algoritmo para calcular multiplicação através de somas consecutivas, para facilitar assumo que os dois fatores da multiplicação são positivos.

### Agora uma dica bônus ...

As linguagens de programação são diferentes umas das outras, mas no fundo a lógica de programação é a mesma (quase sempre). Por exemplo. Na linguagem JAVA, não existe a estrutura REPITA-ATÉ. Mas existe a DO-WHILE, ou seja FAÇA-ENQUANTO. Esta também é pós-testada, mas o teste da condição não é o contrário da WHILE-DO. Pelo motivo óbvio. FAÇA-ENQUANTO (o teste der verdadeiro). Gostou da dica? Agora é com você! Faça o algoritmo para calcular a multiplicação através de somas.

## Capítulo 8 - Loops pré-definidos

No último capítulo você aprendeu a fazer LOOPS. Você descobriu que é possível fazer loops no seu algoritmo através de duas estruturas de repetição ENQUANTO-FAÇA e REPITA-ATÉ e aprendeu a diferença entre estas duas estruturas.

Neste capítulo nós vamos ver a estrutura de LOOP mais utilizada na programação: A estrutura de repetição PARA-FAÇA.

Entender bem esta estrutura determinará se você será um bom ou um mau programador, portanto preste bastante atenção neste capítulo. Releia quantas vezes forem necessárias. No final tem a solução do exercício do último capítulo e um novo exercício para você resolver. Vamos lá?

### O que é um LOOP Pré-definido?

Quando fazemos um algoritmo, muitas vezes já sabemos a quantidade de vezes que um loop deve executar. Por exemplo, some todos os números de 1 a 100. Neste caso, sabemos que o nosso loop deverá ser executado 100 vezes.

O caso mais usado deste tipo de LOOP na programação é quando você deve acessar todos os itens de um vetor, matriz ou lista. (Veremos o que são vetores e matrizes no próximo capítulo)

Por exemplo, uma situação muito comum para programadores. Imagine que você deve enviar um e-mail para todos os clientes cadastrados no seu banco de dados...

Você sabe que tem uma tabela com 3298 clientes no seu banco de dados. Neste caso, você deve fazer um loop de 1 até 3298, e enviar um e-mail para cada cliente.

Entendido o que é um LOOP pré-definido, vejamos qual estrutura de repetição utilizada para este caso.

### A estrutura PARA-FAÇA

Você deve estar imaginando que é possível implementar loop pré-definido utilizando as estruturas de repetição que você aprendeu no capítulo passado.



Sim, é perfeitamente possível! Para isto você precisaria utilizar uma variável que chamamos de "contador".

Esta variável nada mais é do que uma simples variável do tipo inteiro que é responsável por contar quantas iterações (execuções do loop) foram executadas.

Vamos tomar como exemplo o caso que disse anteriormente. Como somar todos os números de 1 a 100.

Um algoritmo com a estrutura ENQUANTO-FAÇA para este problema ficaria assim:

```
Algoritmo "Somar1a100ComEnquanto"
Var
    contador : INTEIRO
    soma : INTEIRO
Inicio
    contador := 1
    soma := 0
    ENQUANTO contador <= 100 FAÇA
        soma := soma + contador
        contador := contador + 1
    FIMENQUANTO
    ESCREVA("A soma de 1 a 100 é: ", soma)
Fimalgoritmo
```

Embora seja possível utilizar estas estruturas de repetição para implementar um loop pré-definido, há uma estrutura criada especificamente para isto. A estrutura de repetição PARA-FAÇA.

O que o PARA-FAÇA faz é justamente implementar um contador implicitamente. Ou seja, as operações de inicializar o contador (`contador := 1`), incrementar o contador (`contador := contador + 1`) e verificar se o LOOP deve continuar (`contador <= 100`) é realizada implicitamente pela estrutura PARA-FAÇA.

O esquema de utilização desta estrutura é assim:

**PARA** <contador> **DE** <valor inicial> **ATE** <valor final> [**PASSO** <valor de incremento>] **FAÇA**

<instruções a serem executadas repetidamente até a <contador> atingir o valor final>

### **FIM-PARA**

A inicialização do contador é realizado implicitamente com o informado na declaração da estrutura. A condição para executar a iteração é que o valor da variável contadora não tenha atingido o <valor final>. E ao final de cada iteração, o valor da variável contadora é incrementado em 1 (ou o valor declarado como PASSO).

Repare que o passo de incremento é opcional, por padrão o contador é incrementado de 1 em 1, mas você pode especificar que quer um outro valor de incremento, por exemplo de 2 em 2 ou de 3 em 3.

Se for usar o incremento padrão de 1 em 1, você pode ignorar o PASSO.

**PARA** <contador> **DE** <valor inicial> **ATE** <valor final> **FAÇA**

<instruções a serem executadas repetidamente até a <contador> atingir o valor final>

### **FIM-PARA**

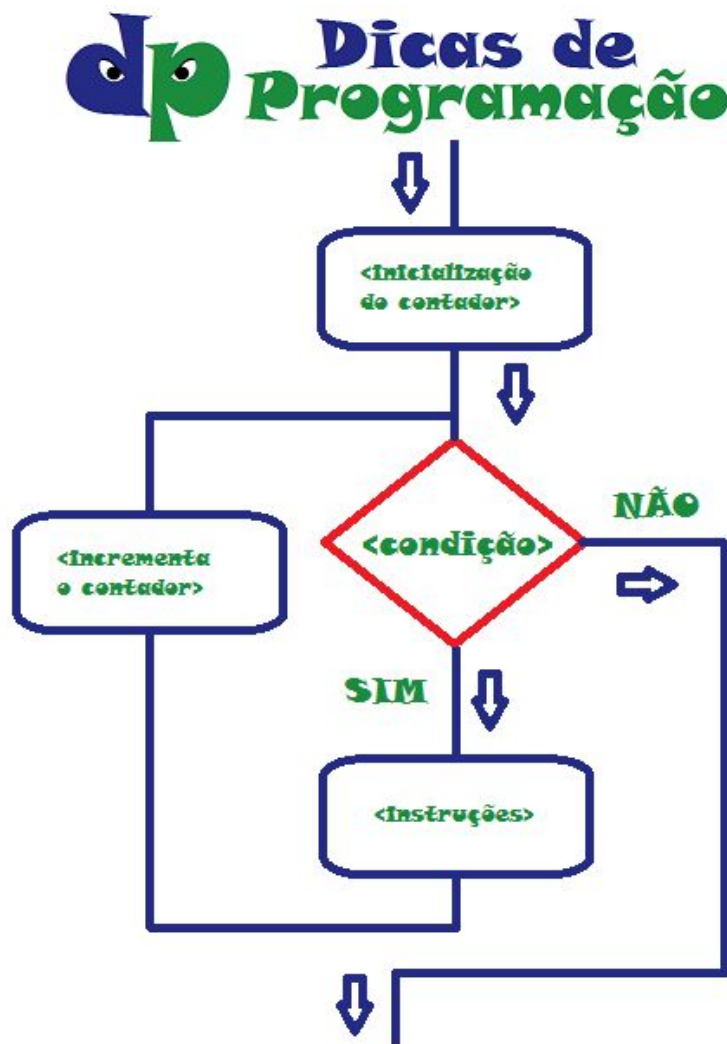
Para o nosso problema de somar todos os números de 1 a 100, um algoritmo com a a estrutura PARA-FAÇA ficaria assim:

```
Algoritmo "Somar1A100ComPara"
Var
    contador : INTEIRO
    soma : INTEIRO
Inicio
    soma := 0
    PARA contador DE 1 ATÉ 100 FAÇA
```

```
soma := soma + contador  
FIMPARA  
    ESCREVA("A soma de 1 a 100 é: ", soma)  
Fimalgoritmo
```

Viu a diferença? No fundo é a mesma coisa, mas para loops pré-definidos a estrutura mais utilizada é a PARA-FAÇA.

Vejamos um fluxograma desta estrutura de repetição:



### Hora de praticar!

Para dar mais um exemplo de LOOP pré-definido. Vamos fazer um algoritmo para resolver um problema matemático: O fatorial de um número.

Se você não sabe, fatorial é a multiplicação de todos os números de 1 até o número que se está calculando. Por exemplo: Fatorial de 5 =  $1 * 2 * 3 * 4 * 5 = 120$ . Fácil né?

Na matemática a notação de fatorial o número e uma exclamação. Por exemplo 5! significa fatorial de 5.

Primeiro vamos fazer um algoritmo utilizando o ENQUANTO.

```
algoritmo "FatorialComENQUANTO"

var
    numero : INTEIRO
    fatorial : INTEIRO
    contador : INTEIRO
inicio

    ESCREVA ("Digite o número para calcular o fatorial: ")
    LEIA (numero)

    fatorial := 1
    contador := 1
    ENQUANTO contador <= numero FAÇA
        fatorial := fatorial * contador
        contador := contador + 1
    FIMENQUANTO

    ESCREVA ("O fatorial de ", numero, " é : ", fatorial)

fimalgoritmo
```

Veja que foi necessário incrementar o contador explicitamente, ou seja, iniciar a variável contador com 1 e incrementar o seu valor no final de cada iteração do LOOP. Com a estrutura de repetição PARA-FAÇA, isso não é necessário. Vejamos agora o mesmo algoritmo implementado com o PARA-FAÇA:

```
algoritmo "FatorialComPARA"

var
    numero : INTEIRO
    fatorial : INTEIRO
    contador : INTEIRO
inicio

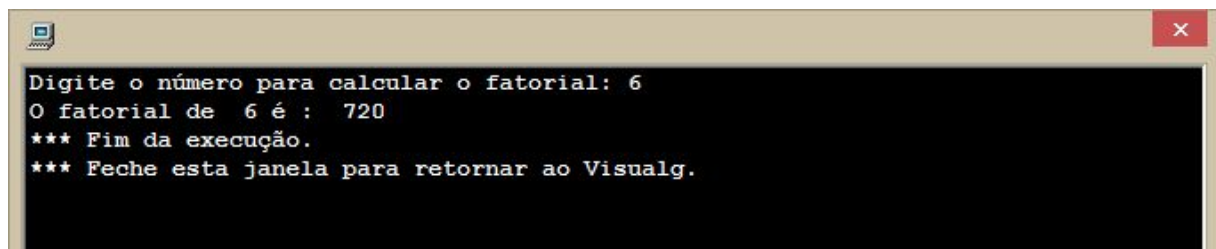
    ESCREVA ("Digite o número para calcular o fatorial: ")
    LEIA (numero)

    fatorial := 1
    PARA contador DE 1 ATE numero FACA
        fatorial := fatorial * contador
    FIMPARA

    ESCREVA ("O fatorial de ", numero, " é : ", fatorial)

finalgoritmo
```

Nesta estrutura, não é necessário incrementar nem inicializar o contador, isso é feito automaticamente. O resultado dos dois algoritmos é o mesmo, veja um exemplo de execução deste algoritmo.



LOOPS podem ser implementados com qualquer estrutura de repetição, porém, em alguns casos uma estrutura se mostra mais adequada que outras, como nesse caso do fatorial a mais adequada é a estrutura PARA. Conhecer essas estruturas de repetição é muito importante para criar programas melhores.

Como eu disse, a estrutura de repetição PARA-FAÇA é muito utilizada para acessar os valores de vetores, matrizes e listas.

### **Novo exercício para você resolver com a estrutura PARA-FAÇA!**

Como sempre digo, lógica de programação só se aprende praticando. Então é a sua vez de tentar resolver um novo problema utilizando algoritmos. O exercício deste capítulo é o seguinte:

**Faça um algoritmo para informar se um determinado número é primo ou não.**

Número primo é todo número que só é divisível por 1 e por ele mesmo sem deixar resto. Exemplos de números primos são: 2, 3, 5, 7, 11, 13, 17 ...

Fácil né!? Dica, você precisará criar um LOOP (de preferência utilizando o PARA) e verificar se o resto das divisões é 0 utilizando o operador **mod**. Por exemplo, a expressão "6 MOD 4" resulta 2, pois é o resto da divisão de 6 por 4.

No final do próximo capítulo você poderá conferir o meu algoritmo para este problema. Assim, você poderá comparar o seu algoritmo com o meu. Mas tente resolver antes heim!

## Capítulo 9 - Vetores e Matrizes (Arrays)

Neste capítulo vamos falar sobre Vetores e Matrizes. Você vai aprender para que serve, como usar e, claro, fazer exercícios para fixar o aprendizado. Ao final deste capítulo você estará craque nesta estrutura de dados tão usada na programação.

### O que são Vetores e Matrizes

**Vetores** e **Matrizes** são estruturas de dados bastante simples que podem nos ajudar muito quando temos um grande número de variáveis do mesmo tipo em um algoritmo.

Imagine o seguinte problema: Você precisa criar um algoritmo que lê o nome e as 4 notas de 50 alunos, calcular a média de cada aluno e informar quais foram aprovados e quais foram reprovados. Conseguiu imaginar quantas variáveis você vai precisar pra fazer este algoritmo?

Muitas né?

Vamos fazer uma continha rápida aqui: são 50 variáveis para armazenar os nomes dos alunos, 200 variáveis para armazenar as 4 notas de cada aluno (4 \* 50) e por fim, 50 variáveis para armazenar as médias de cada aluno.

São 300 variáveis no total, sem contar a quantidade de linhas de código que você vai precisar para ler todos os dados, calcular as médias de cada aluno e apresentar todos os resultados.

Mas eu tenho uma boa notícia pra você! Nós não precisamos criar 300 variáveis! Podemos utilizar **Vetores** e **Matrizes** (também conhecidos como **ARRAYs**)!



Tá bom ... Mas o que são esses tais vetores e matrizes?

**Vetor** (**array** uni-dimensional) é uma variável que armazena várias variáveis do mesmo tipo. No problema apresentado anteriormente, nós podemos utilizar um vetor de 50 posições para armazenar os nomes dos 50 alunos.

**Matriz** (**array** multi-dimensional) é um **vetor** de **vetores**. No nosso problema, imagine uma matriz para armazenar as 4 notas de cada um dos 50 alunos. Ou seja, um vetor de 50 posições, e em cada posição do vetor, há outro vetor com 4 posições. Isso é uma matriz!

Cada item do vetor (ou matriz) é acessado por um número chamado de **índice**. Ou **index** em inglês.

Uma bela forma de pensar software é pensar graficamente... Então vamos imaginar no nosso exemplo dos nomes, notas e médias dos 50 alunos como seriam os vetores e matrizes graficamente para facilitar o entendimento do conceito.



**Vetor de nomes dos alunos**

1	2	3	...	49	50
João	Pedro	Carlos	...	José	Maria

**Matriz das notas dos alunos**

	1	2	3	4
1	9,5	10	8	7,5
2	10	9	9	5,5
3	9	8,5	9,5	7
...	...	...	...	...
49	7	10	10	9
50	7	8,5	5,5	4

Podemos ver na imagem acima que cada posição do vetor é identificado por um número (chamado de **índice**), no caso da matriz são dois números (um na vertical e um na horizontal).

Claro que também pode existir matrizes com mais de duas dimensões, mas não precisa se prender a estes detalhes agora. ;)

No Visualg os vetores são declarados da seguinte maneira:

**< nome da variável> vetor [1..<tamanho>] de <tipo de dados>**

Exemplo:

nomesDosAlunos vetor [1..50] de caractere

E as matrizes assim:

**< nome da variável> vetor [1..<tamanho 1>,1..<tamanho 2>] de <tipo de dados>**

Exemplo:

notas vetor [1..50,1..4] de real

Pronto, agora você já sabe o que são **arrays**. Então vamos ver como implementá-los em um algoritmo.

### Vetores e Matrizes na prática!

Nada como praticar para fixar um aprendizado, concorda?

Continuando com o nosso exemplo, vamos implementar um algoritmo para o cálculo das médias.

Neste algoritmo vamos usar algumas estruturas básicas já apresentadas nas lições anteriores, tais como a estrutura de repetição PARA (capítulo anterior) e a estrutura de decisão SE-ENTÃO-SENÃO (capítulo #5).

*OBS: Neste exemplo vamos reduzir o número de alunos de 50 para 5, para facilitar a visualização do resultado.*

**Preste muita atenção no modo como é criado o Vetor e a Matriz e também a forma como cada posição é acessada, utilizando os contadores.**

```
algoritmo "MediaDe5Alunos"

var

    nomes: vetor [1..5] de caractere
    notas: vetor [1..5,1..4] de real
    medias: vetor [1..5] de real
    contadorLoop1, contadorLoop2: inteiro

inicio

    //Leitura dos nomes e as notas de cada aluno
    PARA contadorLoop1 DE 1 ATE 5 FACA
        ESCREVA("Digite o nome do aluno(a) número ", contadorLoop1, " de 5: ")
        LEIA(nomes[contadorLoop1])
        PARA contadorLoop2 DE 1 ATE 4 FACA
            ESCREVA("Digite a nota ", contadorLoop2, " do aluno(a) ",
nomes[contadorLoop1], ": ")
            LEIA(notas[contadorLoop1, contadorLoop2])
        FIMPARA
    //CÁLCULO DAS MÉDIAS
    medias[contadorLoop1] := (notas[contadorLoop1, 1] + notas[contadorLoop1,
2] + notas[contadorLoop1, 3] + notas[contadorLoop1, 4]) / 4
    FIMPARA
```

```
//APRESENTAÇÃO DOS RESULTADOS
PARA contadorLoop1 DE 1 ATE 5 FAÇA
    SE medias[contadorLoop1] >= 6 ENTAO
        ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi aprovado com as
notas  (", notas[contadorLoop1, 1], ", ", notas[contadorLoop1, 2], ", ",
notas[contadorLoop1, 3], ", ", notas[contadorLoop1, 4], ") e média: ",
medias[contadorLoop1])
    SENAo
        ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi reprovado com as
notas  (", notas[contadorLoop1, 1], ", ", notas[contadorLoop1, 2], ", ",
notas[contadorLoop1, 3], ", ", notas[contadorLoop1, 4], ") e média: ",
medias[contadorLoop1])
    FIMSE
FIMPARA

finalgoritmo
```

Repare que os **arrays** (vetores ou matrizes) aliados à estrutura de repetição PARA é um ótimo recurso para algoritmos que precisam de muitas variáveis do mesmo tipo.

Um resultado do algoritmo acima pode ser observado a seguir:

```
Digite o nome do aluno(a) número 1 de 5: Gustavo
Digite a nota 1 do aluno(a) Gustavo: 9
Digite a nota 2 do aluno(a) Gustavo: 10
Digite a nota 3 do aluno(a) Gustavo: 9,5
Digite a nota 4 do aluno(a) Gustavo: 8
Digite o nome do aluno(a) número 2 de 5: João
Digite a nota 1 do aluno(a) João: 5
Digite a nota 2 do aluno(a) João: 6
Digite a nota 3 do aluno(a) João: 4,5
Digite a nota 4 do aluno(a) João: 7
Digite o nome do aluno(a) número 3 de 5: Pedro
Digite a nota 1 do aluno(a) Pedro: 7
Digite a nota 2 do aluno(a) Pedro: 8,5
Digite a nota 3 do aluno(a) Pedro: 6
```

```
Digite a nota 4 do aluno(a) Pedro: 7
Digite o nome do aluno(a) número 4 de 5: Luciana
Digite a nota 1 do aluno(a) Luciana: 10
Digite a nota 2 do aluno(a) Luciana: 7
Digite a nota 3 do aluno(a) Luciana: 7,5
Digite a nota 4 do aluno(a) Luciana: 8
Digite o nome do aluno(a) número 5 de 5: Augusto
Digite a nota 1 do aluno(a) Augusto: 5
Digite a nota 2 do aluno(a) Augusto: 5,5
Digite a nota 3 do aluno(a) Augusto: 7,5
Digite a nota 4 do aluno(a) Augusto: 6
O aluno(a) Gustavo foi aprovado com as notas ( 9, 10, 9.5, 8) e média: 9.125
O aluno(a) João foi reprovado com as notas ( 5, 6, 4.5, 7) e média: 5.625
O aluno(a) Pedro foi aprovado com as notas ( 7, 8.5, 6, 7) e média: 7.125
O aluno(a) Luciana foi aprovado com as notas ( 10, 7, 7.5, 8) e média: 8.125
O aluno(a) Augusto foi aprovado com as notas ( 5, 5.5, 7.5, 6) e média: 6
*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

Para você que é um iniciante em programação, este algoritmo pode parecer um pouco complexo, mas se prestar atenção, perceberá que os vetores e matrizes podem ser utilizados em muitos problemas. Por exemplo, armazenar os nomes dos funcionários de uma empresa.

Uma coisa importante a se observar é que os arrays são de tamanho fixo, ou seja, eles nascem e morrem com o mesmo tamanho. Se você precisar acrescentar um novo valor em um array e ele já estiver cheio, você deverá criar um novo array maior e realocar os valores do array antigo.

Mas pode ficar tranquilo que existem outras estruturas de dados que crescem dinamicamente, mas isso é assunto para um capítulo futuro ...

### Conclusão

Como você pode perceber neste capítulo, Vetores e Matrizes são, na verdade, a mesma coisa: **ARRAY**

A diferença é que o vetor é um array de apenas 1 dimensão e a matriz é um array de 2 (ou mais) dimensões.

Os arrays também são conhecidos por **variáveis indexadas**.

**Array** é uma das estruturas de dados mais simples que existe e uma das mais utilizadas também. Acho que todas as linguagens de programação têm **arrays**, pelo menos ainda não conheço uma linguagem que não tenha.

É importante falar que estruturas de dados é o assunto seguinte a se aprender, depois de aprender lógica de programação. Existem muitos tipos de estruturas de dados, o array é o mais simples. Conhecer bem as estruturas de dados é o que vai determinar a sua facilidade em aprender qualquer linguagem de programação.

Os índices dos arrays podem mudar dependendo da linguagem, algumas linguagens começam os índices do array com 1 e outras com 0, essa é uma diferença muito comum que encontramos entre linguagens. No caso das linguagens que começam os arrays com o índice 0, o último elemento do array recebe o índice ( $\text{<tamanho do array>} - 1$ ).

### Te desafio a criar um jogo da velha!

Nossa vida é cheia de desafios e eles são muito importantes para evoluirmos e ultrapassar os nossos limites.

Pensando na sua evolução, eu tenho um desafio para você resolver! Neste desafio você poderá utilizar tudo que aprendeu até agora. Inclusive a matriz que você aprendeu neste capítulo!

Quero ver se você aprendeu mesmo!

O desafio é o seguinte:

**Você deverá construir um jogo da velha.**

Simples assim.

Não precisa ser um jogo muito elaborado. Vamos ver alguns requisitos.

1. As jogadas do jogo da velha deverão ser armazenadas numa matriz (3x3) de caractere, chamada "tabuleiro", cada posição desta matriz armazenará um dos valores: " ", "\_", "X" ou "O", onde " " e "\_" são posições *vazias* e "X" e "O" são *jogadas*. Abaixo uma representação gráfica desta matriz.

	1	2	3
1	___	___	___
2	___	___	___
3			

2. A cada jogada o programa deverá mostrar na tela a situação atual do "tabuleiro". Por exemplo:

	1	2	3
1	___	___	___
2	___	_X_	___
3	O		O

3. Terão dois jogadores no jogo. O programa deve solicitar o nome dos dois jogadores antes de começar o jogo.

4. A cada jogada o programa deverá perguntar separadamente as posições horizontal e vertical da jogada, nesta ordem.
5. Quando um jogador vencer o programa deve apresentar imediatamente o vencedor e a situação do "tabuleiro".

Este exercício não é simples, mas com um pouquinho de esforço e persistência tenho certeza que você consegue fazer esse jogo.



## Capítulo 10 - Funções e Procedimentos

Neste capítulo vamos aprender uma forma de melhorar a sua programação. Utilizando **funções** e **procedimentos** nós podemos reaproveitar código, melhorar a leitura dos algoritmos e criar códigos mais limpos e legíveis.

Neste capítulo vamos ver um pouquinho de geometria básica. Só pra lembrar um pouquinho a escola. Mas não se assuste, vai ser fácil.

Vamos lá?

### O que são Funções e Procedimentos

A primeira coisa que você tem que entender é, afinal, que raios são funções e procedimentos?

Bom, já adianto que você já usou procedimentos e nem percebeu!

Lembra quando você quis mostrar algum texto na tela? Você usou o procedimento **ESCREVA** e passou um texto como parâmetro, justamente o texto que você queria que aparecesse na tela.

```
ESCREVA("Olá mundo!")
```

Você saberia mostrar um texto na tela sem usar esse procedimento? Não né.

Outra pergunta: Você saberia fazer um algoritmo para calcular a raiz quadrada de um número? Reflita um pouquinho sobre a complexidade de tal algoritmo. E um algoritmo para gerar um número aleatório? Você saberia fazer?

Imprimir um texto na tela, raiz quadrada, geração de número aleatório, entre outros, são funções e procedimentos clássicos que um programador usa, mas não precisa implementar na unha. Pra quê re-inventar a roda??? Alguém já fez esses algoritmos e a gente apenas usa. O que precisamos é apenas solicitar a execução desses algoritmos dentro do nosso algoritmo.

### Qual a diferença entre função e procedimento?

A única diferença entre uma função (*function*) e um procedimento (*procedure*) é que a função retorna um valor (por exemplo uma função que calcula raiz quadrada retorna um número) e o procedimento não retorna nada (por exemplo o procedimento 'escreva' que já falei).

A figura abaixo exemplifica como acontece a utilização de uma função, o procedimento é a mesma coisa, menos na atribuição do resultado à variável "a".



**Funções** (e **procedimentos**) podem ou não receber parâmetros. No caso da função de raiz quadrada, é necessário passar como parâmetro o número que se

deseja calcular a raiz, o procedimento **ESCREVA**, requer um texto como parâmetro para apresentar na tela do usuário.

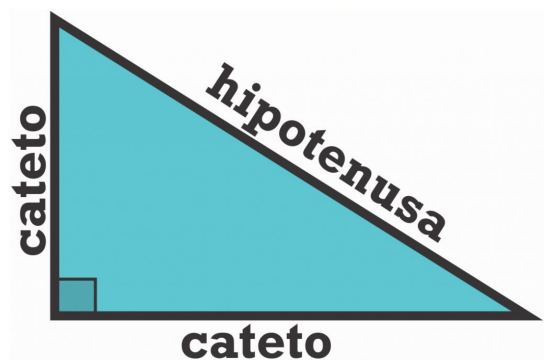
Agora que já sabemos o que são e pra quê servem. Vamos para a prática!

### Hora de praticar: Utilizando funções e procedimentos

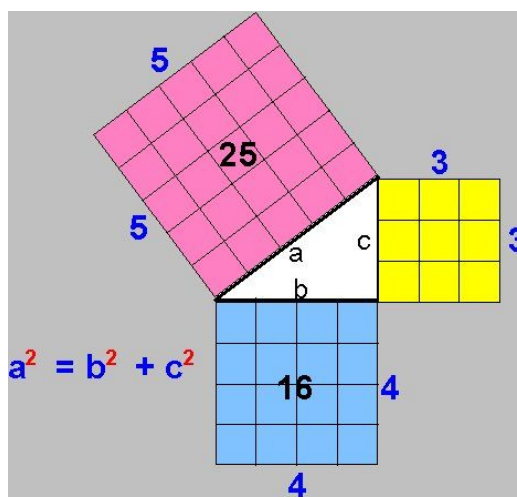
Você lembra como calcular a hipotenusa de um triângulo retângulo?

Primeiro, vou te relembrar o que é um triângulo-retângulo. Um triângulo em que um dos ângulos tem  $90^\circ$ . Ou seja, dois lados do triângulo são perpendiculares entre si. Esses

lados que formam o ângulo de  $90^\circ$  (ou ângulo reto) são chamados de "catetos". E o lado oposto ao ângulo reto é a hipotenusa.



Quando conhecemos o tamanho dos catetos nós conseguimos calcular o tamanho da hipotenusa. Este é o famoso **teorema de Pitágoras** que diz: **A soma dos quadrados dos catetos equivale ao quadrado da hipotenusa**. A imagem abaixo ilustra bem isso.



Então para descobrir o valor da hipotenusa, temos que encontrar a raiz quadrada de  $(b^2 + c^2)$ .

Com base neste cálculo, vamos fazer um algoritmo que solicita ao usuário o valor dos dois catetos, calcula e apresenta na tela o valor da hipotenusa do triângulo retângulo. Para isso precisaremos usar a função RAIZQ do Visualg para calcular a raiz quadrada pra gente.

```
algoritmo "Hipotenusa"
var
    a, b, c : REAL
inicio

    ESCREVA ("Digite o valor do primeiro cateto do triângulo retângulo: ")
    LEIA (b)
    ESCREVA ("Digite o valor do segundo cateto do triângulo retângulo: ")
    LEIA (c)

    a := RAIZQ ( b*b + c*c )//Cálculo da hipotenusa utilizando a FUNÇÃO RAIZQ,

    ESCREVA ("O valor da hipotenusa é: ", a)

fimalgoritmo
```

Observe que utilizamos a função RAIZQ para calcular a raiz quadrada do valor que passamos como parâmetro (valor entre parênteses) " $b*b + c*c$ ", o valor retornado por essa função armazenamos na variável "a".

## Como criar as suas próprias funções e procedimentos

Você também pode criar as suas próprias funções e procedimentos. Entre as vantagens de criar as próprias funções e procedimentos cito duas, melhora a

legibilidade do código, tirando complexidades de dentro do fluxo principal do seu algoritmo e remove repetição de código.

Abaixo a sintaxe para criação das suas próprias funções e procedimentos no Visualg.

```
funcao <nome-de-função> [(<seqüência-de-declarações-de-parâmetros>)] : <tipo-de-dado>
// Seção de Declarações Internas
inicio
// Seção de Comandos
fimfuncao
```

```
procedimento <nome-de-procedimento> [(<seqüência-de-declarações-de-parâmetros>)]
// Seção de Declarações Internas
inicio
// Seção de Comandos
fimprocedimento
```

Vamos criar e usar uma função pra praticar. Vamos criar uma função que recebe um número inteiro e retorna o fatorial deste número.

Fatorial é a multiplicação de todos os números entre 1 e o número especificado.

Exemplo: Fatorial de 5 (ou 5!) corresponde a:  $1 * 2 * 3 * 4 * 5 = 120$

Então vamos ver como ficaria esta função.

```
funcao calculaFatorial(numero: inteiro): inteiro
var
    fatorial: inteiro
    contador: inteiro
inicio
    fatorial <- 1
    ENQUANTO numero > 1 FACA
        fatorial <- fatorial * numero
        numero <- numero - 1
```

```
FIMENQUANTO
    retorne fatorial
fimfuncao
```

O fluxo principal do nosso Algoritmo poderia ser assim.

```
ESCREVA("Informe o número para o cálculo do Fatorial: ")
LEIA(numeroParaFatorial)
ESCREVA("O fatorial de ", numeroParaFatorial, " é: ",
calculaFatorial(numeroParaFatorial))
```

Esse é o algoritmo completo, com a função e o fluxo principal.

```
algoritmo "Cacula Fatorial"
var

    numeroParaFatorial: inteiro

funcao calculaFatorial(numero: inteiro): inteiro
var
    fatorial: inteiro
    contador: inteiro
inicio
    fatorial <- 1
    ENQUANTO numero > 1 FACA
        fatorial <- fatorial * numero
        numero <- numero - 1
    FIMENQUANTO
    retorne fatorial
fimfuncao

inicio

    ESCREVA("Informe o número para o cálculo do Fatorial: ")
    LEIA(numeroParaFatorial)
    ESCREVA("O fatorial de ", numeroParaFatorial, " é: ",
calculaFatorial(numeroParaFatorial))
```

fimalgoritmo

### Resumindo

Vimos neste capítulo que **Funções** e **procedimentos** são "subalgoritmos" que podem ser chamados dentro de outros algoritmos.

São utilizados com muita frequência em desenvolvimento de softwares. Existem vários benefícios como: evita duplicação de código quando precisamos executar a mesma operação várias vezes, deixa o entendimento do algoritmo mais intuitivo, pois tiramos a parte complexa do código do fluxo principal do algoritmo, etc.

**Importante:** em linguagens orientada a objeto como java, C++ e C#, funções e procedimentos são chamados de **MÉTODOS**. Mais por uma questão de conceito de Orientação a Objetos, mas no fundo é a mesma coisa, podem receber parâmetros e retornam ou não um resultado.

## Solução do exercício do capítulo 2

Se não conseguiu fazê-lo, não tem problema. Eu pedi para você solicitar as 4 notas do usuário, calcular a média e apresentar na tela. Neste capítulo você aprendeu sobre variáveis e os tipos de dados. Para resolver este exercício você precisará criar 5 variáveis do tipo real, 4 variáveis para armazenar as 4 notas e uma para armazenar a média.

Em seguida nós devemos solicitar ao usuário que digite as notas e armazená-las nas respectivas variáveis.

O passo seguinte é o cálculo da média, ou seja, a soma das 4 notas dividido por 4. Repare que precisamos colocar as somas entre parênteses, pois os operadores de multiplicação e divisão têm precedência quanto aos operadores de soma e subtração. Você vai aprender um pouco mais sobre os operadores no próximo capítulo.

O resultado do cálculo é armazenado na variável "media". Por fim, apresentamos a média na tela para o usuário.

Aqui está o meu algoritmo:

```
algoritmo "MédiaAnoLetivo"
var
    nota1, nota2, nota3, nota4, media : real
inicio
    escreva("Digite a primeira nota para o calculo da media: ")
    leia(nota1)
    escreva("Digite a segunda nota para o calculo da media: ")
    leia(nota2)
    escreva("Digite a terceira nota para o calculo da media: ")
    leia(nota3)
    escreva("Digite a quarta nota para o calculo da media: ")
    leia(nota4)

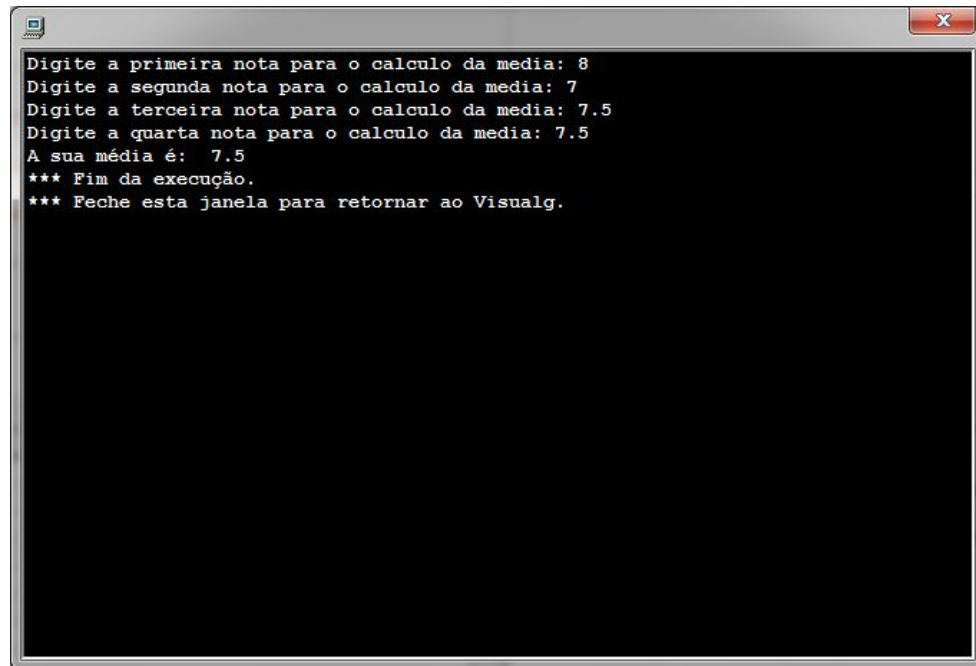
    media <- ( nota1 + nota2 + nota3 + nota4 ) / 4

    escreva("A sua média é: ", media)
```



fimalgoritmo

Apresento abaixo o resultado da execução deste algoritmo.



```
Digite a primeira nota para o calculo da media: 8
Digite a segunda nota para o calculo da media: 7
Digite a terceira nota para o calculo da media: 7.5
Digite a quarta nota para o calculo da media: 7.5
A sua média é: 7.5
*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

## Solução do exercício do capítulo 5

No final do capítulo 5 eu pedi pra você tentar resolver um exercício de lógica para verificar se um aluno foi aprovado ou reprovado no final do ano.

Você fez? Espero que sim! Teve alguma dificuldade? Bom, abaixo eu mostro como eu escrevi um algoritmo para resolver esse exercício. Compare com o que você fez. Se o seu não deu certo, continue lendo que eu explico cada parte do algoritmo.

Esse é o algoritmo:

```
algoritmo "AprovacaoFinalDeAno"
var
    nota1, nota2, nota3, nota4, media: real
inicio

    escreva("Informe a nota (de 0 a 10) do primeiro bimestre: ")
    leia(nota1)
    escreva("Informe a nota (de 0 a 10) do segundo bimestre: ")
    leia(nota2)
    escreva("Informe a nota (de 0 a 10) do terceiro bimestre: ")
    leia(nota3)
    escreva("Informe a nota (de 0 a 10) do quarto bimestre: ")
    leia(nota4)

    media := (nota1 + nota2 + nota3 + nota4) / 4

    escreval("Sua média foi: ", media)

    se media >= 6 entao
        escreva("Você foi APROVADO!")
    senao
        escreva("Você foi REPROVADO!")
    fimse

fimalgoritmo
```

Entendendo o algoritmo. Primeiro eu declarei 5 variáveis do tipo REAL. Elas têm que ser do tipo REAL porque as notas podem ter valores decimais, por exemplo 5.5.

Depois eu escrevi na tela "Digite a nota (de 0 a 10) do primeiro bimestre: " e armazenei na variável nota1 o valor que o usuário digitou. Fiz o mesmo para as outras 3 notas.

Na sequência e calculei a média das 4 notas e armazenei o resultado na variável "media". Importante colocar o parênteses para somar as notas ANTES de dividir por 4.

Agora que vem a parte da decisão, o SE-ENTÃO-SENÃO.

Eu verifiquei se a média é MAIOR OU IGUAL a 6. Se SIM ENTÃO imprimi na tela a mensagem informando que o aluno foi aprovado. SENÃO imprimi a mensagem informando que o aluno foi reprovado.

Veja abaixo o resultado da execução do algoritmo no Visualg, quando a média era menor que 6 e quando foi maior.

Viu como foi simples? Se você teve dificuldades para resolver, não se preocupe. No início parece difícil mesmo. Mas como sempre digo, é preciso praticar!

Se conseguiu resolver sem dificuldades ótimo, mas continue praticando.

## Solução do exercício do capítulo 6

Espero que você tenha tentado fazer esse exercício sozinho. Afinal, treinar é muito importante.

Eu escrevi um algoritmo com a solução deste exercício usando a estrutura ESCOLHA-CASO. Dê uma olhada:

```
algoritmo "Posição da letra no alfabeto"
var
    letra : CARACTERE
    posicao : INTEIRO
inicio

    ESCREVA("Digite uma letra: ")
    LEIA(letra)

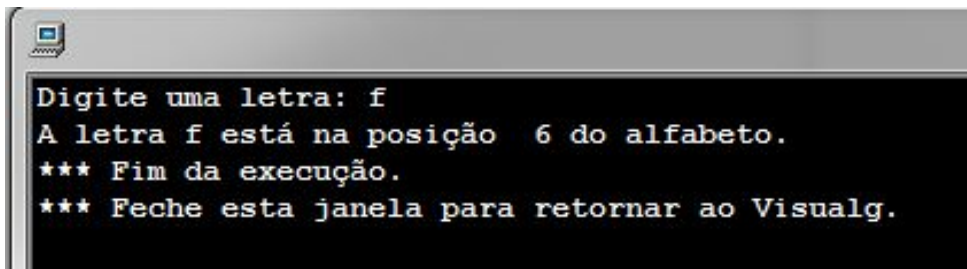
    ESCOLHA letra
        CASO "a"
            posicao := 1
        CASO "b"
            posicao := 2
        CASO "c"
            posicao := 3
        CASO "d"
            posicao := 4
        CASO "e"
            posicao := 5
        CASO "f"
            posicao := 6
        CASO "g"
            posicao := 7
        CASO "h"
            posicao := 8
        CASO "i"
            posicao := 9
        CASO "j"
            posicao := 10
        CASO "k"
            posicao := 11
        CASO "l"
```

```
        posicao := 12
CASO "m"
        posicao := 13
CASO "n"
        posicao := 14
CASO "o"
        posicao := 15
CASO "p"
        posicao := 16
CASO "q"
        posicao := 17
CASO "r"
        posicao := 18
CASO "s"
        posicao := 19
CASO "t"
        posicao := 20
CASO "u"
        posicao := 21
CASO "v"
        posicao := 22
CASO "w"
        posicao := 23
CASO "x"
        posicao := 24
CASO "y"
        posicao := 25
CASO "z"
        posicao := 26
FIMESCOLHA
```

```
        ESCREVA("A letra ", letra, " está na posição ", posicao, " do alfabeto.")
```

```
fimalgoritmo
```

Olha um resultado da execução deste algoritmo:



É possível implementar um algoritmo com a estrutura SE-ENTÃO-SENÃO, mas ficaria bem maior. Veja só o início deste algoritmo:

```
algoritmo "Posição da letra no alfabeto com SE"
var
    letra : CARACTERE
    posicao : INTEIRO
inicio

    ESCREVA("Digite uma letra: ")
    LEIA(letra)

    SE letra = "a" ENTÃO
        posicao := 1
    SENÃO
        SE letra = "b" ENTÃO
            posicao := 2
        SENÃO
            SE letra = "c" ENTÃO
                posicao := 3
            SENÃO
                SE letra = "d" ENTÃO
                    posicao := 4
                SENÃO
                    SE letra = "e" ENTÃO
                        posicao := 5
                    SENÃO
                        SE ....
                        .....
                    FIMSE
                FIMSE
            FIMSE
        FIMSE
    FIMSE
FIMSE
```

```
    ESCREVA("A letra ", letra, " está na posição ", posicao, " do  
alfabeto.")  
  
finalgoritmo
```

Bom agora eu tenho uma surpresa pra você! Se você acompanhou esse exercício até aqui e está gostando do e-book, eu vou te ensinar como fazer todo o trabalho dessa estrutura ESCOLHA-CASO com apenas UMA LINHA de código e sem usar nenhuma estrutura de controle de fluxo!

Isso mesmo, um algoritmo que diz a ordem da letra no alfabeto sem usar nenhuma estrutura de controle de fluxo como você aprendeu nas duas últimas lições. Ficou curioso? A malandragem é a seguinte...

Na computação, todos os caracteres tem um correspondente numérico para que este caractere possa ser armazenado na forma de bits.

Existe uma tabela chamada **Tabela ASCII** para sabermos qual o número de uma letra. E as letras do alfabeto estão em sequência nesta tabela.

Veja abaixo uma parte da tabela ASCII e identifique o valor numérico do caractere "a".

Número	Caractere	Número	Caractere	Número	Caractere	Número	Caractere	Número	Caractere	Número	Caractere
32	ESPAÇO	48	0	64	@	80	P	96	·	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	96	_	111	o	127	DELETE

Viu que o valor do caractere "a" é 97 e que as outras letras estão na sequência? b = 98, c = 99, d = 100, ...

Agora ficou fácil, só precisamos descobrir o valor da letra que o usuário digitou e subtrair 96. Certo?

Para descobrir o valor ASCII de um caractere no Visualg, podemos utilizar a *função ASC*, passando como parâmetro a letra que o usuário digitou.

**Importante!** Vamos falar mais sobre **funções** e **procedimento** no capítulo 10, não se preocupe. Por hora só saiba que uma função executa uma tarefa pra gente. (Talvez a função ASC use a estrutura ESCOLHA-CASO internamente para retornar o número ASCII da letra.)

A função `ASC(caracter)` retorna o número da tabela ASCII da letra que passamos como parâmetro.

Logo, o nosso algoritmo ficaria assim:

```
algoritmo "Posição da letra no alfabeto"
var
```



```
letra : CARACTERE
posicao : INTEIRO
inicio

    ESCREVA("Digite uma letra: ")
    LEIA(letra)

    posicao := ASC(letra) - 96

    ESCREVA("A letra ", letra, " está na posição ", posicao, " do
alfabeto.")

finalgoritmo
```

O resultado é o mesmo do algoritmo que usa a estrutura ESCOLHA-CASO ou SE-ENTÃO-SENÃO.

## Solução do exercício do capítulo 7

No final do capítulo 7 eu pedi pra você resolver um exercício criando um algoritmo capaz de fazer multiplicação de dois números positivo.

Espero que você tenha tentado fazer sozinho heim! Se não fez, tente fazer primeiro pra depois olhar a resposta que apresento abaixo.

Eu mostrei esse algoritmo no primeiro capítulo deste e-book. Lembra? Talvez naquele momento você não tenha compreendido direito, mas agora você já tem o conhecimento mínimo para fazer um algoritmo de multiplicação. Aqui está o meu algoritmo de multiplicação entre números positivos:

```
algoritmo "Multiplicação"
var
    numero1, numero2, resultado, contador: INTEIRO
inicio
    ESCREVA("Informe o primeiro número: ")
    LEIA(numero1)
    ESCREVA("Informe o segundo número: ")
    LEIA(numero2)

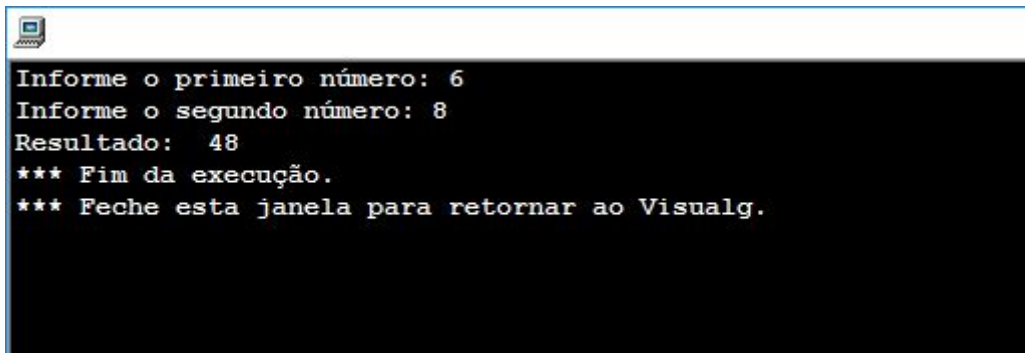
    contador <- 0
    resultado <- 0

    ENQUANTO ( contador < numero2 ) FACA
        resultado <- resultado + numero1
        contador <- contador + 1
    FIMENQUANTO

    ESCREVA("Resultado: ", resultado)

fimalgoritmo
```

Aqui um resultado da execução deste algoritmo.



```
Informe o primeiro número: 6
Informe o segundo número: 8
Resultado: 48
*** Fim da execução.
*** Feche esta janela para retornar ao Visualg.
```

Nesse algoritmo nós definimos como realizar uma multiplicação somando o *número1* a quantidade de vezes do *número2*. Exatamente como aprendemos na escola! A gente controla a execução do loop com a variável *contador*. Nós também podemos implementar esse algoritmo utilizando a estrutura PARA-FAÇA, que aprendemos neste capítulo.

Com a estrutura PARA-FAÇA, esse algoritmo ficaria assim:

```
algoritmo "MultiplicaçãoComParaFaca"
var
    numero1, numero2, resultado, contador: INTEIRO
inicio
    ESCREVA("Informe o primeiro número: ")
    LEIA(numero1)
    ESCREVA("Informe o segundo número: ")
    LEIA(numero2)

    resultado <- 0

    PARA contador DE 1 ATE numero2 FAÇA
        resultado <- resultado + numero1
    FIMPARA

    ESCREVA("Resultado: ", resultado)
```

fimalgoritmo

## Solução do exercício do capítulo 8

### Algoritmo de identificação de números primos

No final do capítulo 8 deste e-book de lógica de programação, eu pedi pra você resolver um exercício.

**Fazer um algoritmo para dizer se um determinado número é primo ou não.**

E aí, conseguiu fazer? Espero que você tenha tentado e conseguido fazer sozinho!

Se não conseguiu, tudo bem, com a prática você vai ficando craque na lógica de programação.

O problema é simples, como eu disse no capítulo anterior, um número primo só pode ser divisível (resto = 0) por 1 e por ele mesmo, ou seja, se ele for divisível por qualquer outro número entre 2 e ele mesmo menos 1, ele não é primo. Sacou?

Abaixo você vai ver o algoritmo que eu fiz para este problema.

```
Algoritmo "NumeroPrimo"
Var
    contador : INTEIRO
    numero : INTEIRO
    eprimo : LOGICO
Inicio
    ESCREVA("Informe um número para verificar se ele é primo: ")
    LEIA(numero)
    eprimo := VERDADEIRO
    PARA contador DE 2 ATÉ numero-1 FAÇA
        SE (numero MOD contador) = 0 ENTAO
```

```
        eprimo := FALSO
    FIMSE
FIMPARA
SE eprimo = VERDADEIRO ENTAO
    ESCREVA("O número ", numero, " é primo!")
SENAO
    ESCREVA("O número ", numero, " NÃO é primo!")
FIMSE
Fimalgoritmo
```

Neste algoritmo eu faço um *loop* de 2 até o número imediatamente anterior ao número que estou verificando se é primo. Por quê?

Bom, eu já expliquei que um número primo só é divisível (resto 0) por 1 e por ele mesmo. Ele não deve ser divisível por qualquer outro número.

Então, caso algum número entre 2 e "numero-1" seja capaz de dividir o número verificado com resto zero (SE (numero MOD contador) = 0 ENTÃO ...), significa que ele **não** é primo (eprimo = FALSO).

Aqui um resultado da execução deste algoritmo.

```
Início da execução
Informe um número para verificar se ele é primo: 53
O número 53 é primo!
Fim da execução.
```

O que você achou? Gostou da minha resolução? O seu algoritmo pode ter sido diferente. Não tem problema. Há várias formas de se fazer algoritmos. Não precisa estar igual o meu. Só precisa funcionar corretamente. ;)

Aliás essa é a beleza da lógica de programação.

## Solução do exercício do capítulo 9

### Algoritmo do jogo da velha

No capítulo 9 te desafiei a fazer um algoritmo do jogo da velha.

Não é um algoritmo facinho, mas já estamos chegando ao final deste e-book e sei que você tem capacidade de criar este algoritmo sozinho.

Mas você tinha que quebrar a cabeça um pouquinho. Neste exercício, que passei como um desafio para você, é necessário utilizar várias estruturas, operadores, variáveis, etc. Tudo que já aprendemos nas lições anteriores.

Então vamos ao meu algoritmo do jogo da velha, mas antes vamos lembrar as regras:

1 - As jogadas do jogo da velha deverão ser armazenadas numa matriz (3x3) de caractere, chamada "tabuleiro", cada posição desta matriz armazenará um dos valores: " ", "\_", "X" ou "O". Abaixo uma representação gráfica desta matriz.

	1	2	3		
1	__		__		__
2	__		__		__
3					

2 - A cada jogada o programa deverá mostrar na tela a situação atual do "tabuleiro". Por exemplo:

```
1   2   3
1  _|_|_
2  _|_X_|_
3  0|_|0
```

3 - Terão dois jogadores no jogo. O programa deve solicitar o nome dos dois jogadores antes de começar o jogo. A cada jogada o programa deverá perguntar separadamente as posições horizontal e vertical da jogada, nesta ordem.

4 - Quando um jogador vencer o jogo, o programa deve apresentar imediatamente o vencedor e a situação do "tabuleiro".

Abaixo você encontra o meu algoritmo do jogo, você pode copiar e colocar o algoritmo no VisuAlg.

Para baixar o Visualg Acesse:

<https://dicasdeprogramacao.com.br/download-visualg/>

Veja o algoritmo, entenda, [execute-o](#), observe porque usei cada estrutura PARA-FAÇA, REPIRA-ATÉ, SE-ENTÃO-SENÃO, cada variável, operadores lógicos (E e OU) etc.

```
algoritmo "JogoDaVelha"
var
    tabuleiro: vetor[1..3,1..3] de caractere
    nomeJogador1, nomeJogador2, jogadorAtual, vencedor : caractere
    linhaJogada, colunaJogada, i, j : inteiro
inicio
    escreval("###Jogo da velha###")

    //Inicialização do tabuleiro com "_" nas linhas 1 e 2 e " " na terceira linha
    //i é a linha e j é a coluna.
    para j de 1 ate 3 faca
        para i de 1 ate 2 faca
            tabuleiro[i,j] := "_"
```



```
fimpara
    tabuleiro[3,j] := " "
fimpara

escreva("Informe o nome do(a) primeiro(a) jogador(a): ")
leia(nomeJogador1)
escreva("Informe o nome do(a) segundo(a) jogador(a): ")
leia(nomeJogador2)

escreval("Vamos começar o jogo.")

jogadorAtual := nomeJogador1
vencedor := ""

repita
    //Apresenta a situação atual do tabuleiro
    escreval("Neste momento o tabuleiro está assim:")
    escreval("  1   2   3")
    escreval("1 _", tabuleiro[1,1], "_|_", tabuleiro[1,2], "_|_", tabuleiro[1,3],
" _")
    escreval("2 _", tabuleiro[2,1], "_|_", tabuleiro[2,2], "_|_", tabuleiro[2,3],
" _")
    escreval("3  ", tabuleiro[3,1], " | ", tabuleiro[3,2], " | ", tabuleiro[3,3], "
")

    escreval("É a vez do(a) jogador(a): ", jogadorAtual)

repita
    repita
        escreva("Informe o número da linha da sua jogada: ")
        leia(linhaJogada)
        //Valida se o usuário digitou um valor válido
        se (linhaJogada < 1) ou (linhaJogada > 3) entao
            escreval("A linha deve ser entre 1 e 3")
        fimse
    ate (linhaJogada >= 1) e (linhaJogada <= 3)
    repita
        escreva("Informe o número da coluna da sua jogada: ")
```

```
        leia(colunaJogada)
        //Valida se o usuário digitou um valor válido
        se ((colunaJogada < 1) ou (colunaJogada > 3)) entao
            escreval("A coluna deve ser entre 1 e 3")
        fimse
    ate ((colunaJogada >= 1) e (colunaJogada <= 3))

    //Valida se a posição jogada á está preenchida
        se (tabuleiro[linhaJogada,colunaJogada] <> "_") e
(tabuleiro[linhaJogada,colunaJogada] <> " ") entao
            escreval("A posição ", linhaJogada, ", ", colunaJogada, " já está
preenchida.")
        fimse
    ate (tabuleiro[linhaJogada,colunaJogada] = "_") ou
(tabuleiro[linhaJogada,colunaJogada] = " ")

    se jogadorAtual = nomeJogador1 entao
        tabuleiro[linhaJogada,colunaJogada] := "X"
        jogadorAtual := nomeJogador2
    senao
        tabuleiro[linhaJogada,colunaJogada] := "O"
        jogadorAtual := nomeJogador1
    fimse

    //Valida se alguém ganhou o jogo
    para j de 1 ate 3 faca
        //Verifica as colunas
        //_X_|_O_|_X_
        //_X_|_O_|_X_
        //_X_|_O_|_X_
        se ((tabuleiro[1,j] = "X") ou (tabuleiro[1,j] = "O")) e (tabuleiro[1,j] =
tabuleiro[2,j]) e (tabuleiro[2,j] = tabuleiro[3,j]) entao
            vencedor := tabuleiro[1,j]
        fimse
    fimpara

    para i de 1 ate 3 faca
        //Verifica as linhas
        //_X_|_X_|_X_
```

```
//_o_|_o_|_o_
// x | x | x
    se ((tabuleiro[i,1] = "X") ou (tabuleiro[i,1] = "O")) e (tabuleiro[i,1] =
tabuleiro[i,2]) e (tabuleiro[i,2] = tabuleiro[i,3]) entao
        vencedor := tabuleiro[i,1]
    fimse
fimpara

//Verifica as diagonais
//_X_|_|_X_
//_|_X_|_|
// x |   | x
    se (((tabuleiro[2,2] = "X") ou (tabuleiro[2,2] = "O")) e ((tabuleiro[1,1] =
tabuleiro[2,2]) e (tabuleiro[2,2] = tabuleiro[3,3])) ou ((tabuleiro[3,1] =
tabuleiro[2,2]) e (tabuleiro[2,2] = tabuleiro[1,3]))) entao
        vencedor := tabuleiro[2,2]
    fimse

//Verifica se deu velha
    se ((vencedor <> "") e (tabuleiro[1,1] <> "_") e (tabuleiro[1,2] <> "_") e
(tabuleiro[1,3] <> "_") e (tabuleiro[2,1] <> "_") e (tabuleiro[2,2] <> "_") e
(tabuleiro[2,3] <> "_") e (tabuleiro[3,1] <> " ") e (tabuleiro[3,2] <> " ") e
(tabuleiro[3,3] <> " ")) entao
        vencedor := "V"
    fimse

ate vencedor <> ""

//Apresenta a situação atual do tabuleiro
escreval("Neste momento o tabuleiro está assim:")
escreval("  1   2   3")
escreval("1 _", tabuleiro[1,1], "_|_", tabuleiro[1,2], "_|_", tabuleiro[1,3], "_")
escreval("2 _", tabuleiro[2,1], "_|_", tabuleiro[2,2], "_|_", tabuleiro[2,3], "_")
escreval("3  ", tabuleiro[3,1], " | ", tabuleiro[3,2], " | ", tabuleiro[3,3], " ")

se vencedor = "X" entao
    escreva("O vencedor do jogo foi: ", nomeJogador1)
```

```
fimse
se vencedor = "O" entao
    escreva("O vencedor do jogo foi: ", nomeJogador2)
fimse
se vencedor = "V" entao
    escreva("O jogo deu Velha!")
fimse

finalgoritmo
```

Se tiver dúvida, acesse as lições anteriores, onde falo sobre cada um desses assuntos.

## Meu agradecimento

Não queria terminar este e-book sem agradecer a você por ter acompanhado este texto e por ter saído da sua zona de conforto para se dedicar a aprender uma coisa nova. Saiba que o mundo precisa de mais pessoas como você!

Espero que você utilize este conhecimento para o bem e contribua com o mundo criando tecnologia que melhore a vida das pessoas.

Obrigado!

Gustavo