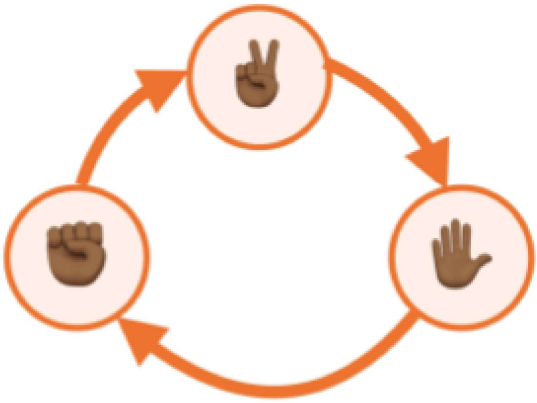
Programação para Todos

Do ZERO ao Básico

- LINGUAGEM ACESSÍVEL
- EXERCÍCIOS RESOLVIDOS
- PRÁTICAS COM JAVASCRIPT
- TUTORIAIS COM REPLIT



ESCRITO POR DANILO BORGES DA SILVA

Danilo Borges da Silva

Programação para todos

Do ZERO ao Básico

EdUESPI

Universidade Estadual do Piauí
Rua João Cabral • n. 2231 • Bairro Pirajá• Teresina-PI

Marcelo de Sousa Neto **Editor**
Ana Christina de Sousa Damasceno **Revisão**
Editora e Gráfica - UESPI **E-book**

S586p Silva, Danilo Borges da.

Programação para todos: do zero ao básico [recurso eletrônico] / Danilo Borges da Silva. - Teresina: EdUESPI, 2022.
E-book

ISBN: 978-65-88108-59-8

1. Programação - Computação. 2. Programas de computador. 3. Linguagem de programação. 4. Pensamento computacional. I. Título.

CDD: 005

Ficha Catalográfica elaborada pelo Serviço de Catalogação da Universidade Estadual do Piauí -UESPI
Ana Angélica P. Teixeira (Bibliotecária) CRB 3a/1217

Editora da Universidade Estadual do Piauí - EdUESPI
UESPI (*Campus Poeta Torquato Neto*)
Rua João Cabral, 2231 • Bairro Pirajá • Teresina-PI

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

Apresentação

Este livro é um presente aos alunos do curso de extensão promovido pela Universidade Estadual do Piauí (UESPI), que teve como coordenador o Professor Me. Danilo Borges da Silva, que pertence ao quadro efetivo do curso de Ciência da Computação desta universidade.

A ideia de produzir este livro surgiu de um questionário de *feedback* preenchido pelos alunos da primeira edição do curso: Programação para Todos: do Zero ao Básico. Esse curso foi ofertado pela primeira vez de forma pública e gratuita em 2021, na modalidade remota (on-line), e obteve sucesso em quantidade de inscrições.

Durante a realização da segunda edição do curso, em 2022, este livro começou a ser produzido e disponibilizado em capítulos separados ao passo que as aulas avançavam. A crítica dos alunos foi determinante para que o material fosse revisado e ganhasse a forma deste livro. Todo o conteúdo do curso foi incorporado nesta obra, que passa agora a ser uma referência para os primeiros passos de todos/as que querem aprender a programar.

Este livro é dividido em três partes:

I Pensando de Forma Computacional;
II Conhecendo os Comandos Básicos; e
III Alterando o Fluxo do Programa.

Na Parte I: Pensando de Forma Computacional, o leitor é convidado a ver como a computação está imersa no dia a dia e como começar a trabalhar o pensamento computacional baseado na lógica matemática. Nessa parte são apresentadas as formas de representar algoritmos: descrição narrativa, fluxograma e pseudocódigo.

Na Parte II: Conhecendo os Comandos Básicos, são apresentadas as ferramentas base para começar a tirar os algoritmos do papel e transformá-los em programas de computador. Serão conhecidos os tipos de dados armazenados e lidos pelo programa e como fazer este procedimento de leitura e de escrita de dados. Nessa parte, será utilizada a linguagem de programação *javascript* para execução dos programas dentro de um ambiente de programação na nuvem, chamado Replit.

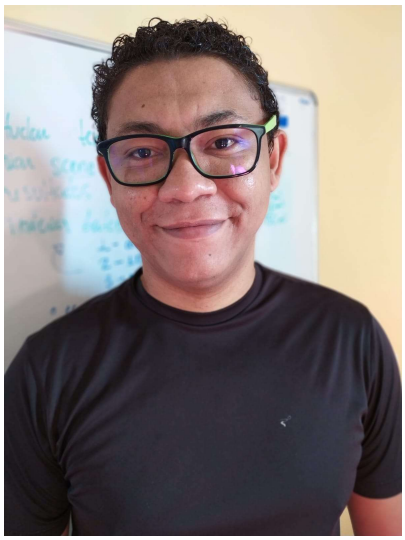
Na Parte III: Alterando o Fluxo do Programa, serão apresentadas as formas básicas de se alterar o fluxo de um programa, como: comandos condicionais e comandos de repetição.

Elementos essenciais para promover o dinamismo da solução computacional e interação do usuário no programa. Os fluxos são apresentados de forma bem natural para que o leitor consiga abstrair seu uso.

Neste livro o autor conversa com o leitor durante o percurso de leitura e o guia na jornada da programação. Apresenta também exercícios e soluções. As videoaulas[1] do curso podem ser utilizadas para um melhor aproveitamento do leitor.

Boa leitura!

Sobre o Autor



Danilo Borges da Silva nasceu em 1989, na cidade de Floriano (PI), atualmente reside em Parnaíba (PI). Filho de professora e de mecânico de máquinas pesadas, observou na educação o potencial para transformar a sociedade. Danilo formou-se em Ciência da Computação pela Universidade Estadual do Piauí (UESPI) e em Licenciatura Plena em Matemática pela Universidade Federal do Piauí (UFPI). Em 2014, concluiu seu mestrado em Ciência da Computação pela Universidade Federal de Ceará (UFC). Além de amar programar, tem como grande paixão a docência. Desde outubro de 2018 exerce a profissão de professor efetivo, com dedicação exclusiva, na Universidade Estadual do Piauí (UESPI). Como meio de prestar serviços à comunidade realiza anualmente projetos, e cursos, de extensão na modalidade on-line e gratuitos.
O professor Danilo nunca havia pensado escrever um livro, apesar de ter nascido no dia em que se comemora o dia do livro, 18 de abril. A oportunidade veio com o curso de extensão e com a expectativa de que o conhecimento sobre computação chegue a mais pessoas, vos apresenta esta obra que dialoga entre a programação e a sua aplicabilidade prática.

[1] <https://bit.ly/dozeroaobasico>

Conteúdo

Pensando de Forma Computacional

1. Iniciando a Jornada da Programação

Neste capítulo você aprenderá a:

- Conhecer quais os principais requisitos para se tornar um programador;
- Estruturar a lógica de pensamento a partir de premissas obtendo, assim, conclusões.

1.1 Programação e seus benefícios

Programar é uma atividade que pode trazer benefícios para qualquer pessoa que possa atuar no ato de pensar de forma lógica auxiliando, assim, no processo de resolução de problemas (MARTINS et al., 2022). Embora, o uso da palavra remeta, para grande maioria das pessoas, se organizar, "Se eu me programar dá certo", vê-se algo bastante semelhante com a ideia de programação de computadores. Quando falamos em nos programarmos estamos fazendo o uso da seguinte lógica: eu irei me organizar para fazer tal coisa em um determinado dia, horário ou período do dia. A organização envolve um dos elementos principais da programação: a lógica!

Para dar um bom *start* (começo) vamos falar de alguns dos motivos para se aprender a programar (ALVES, 2018):

- *Alfabetização.* A programação está sendo introduzida desde o início da educação, ainda na alfabetização como disciplina básica. Desde 2001, existem relatos de trabalhos onde o ensino de programação é realizado ao longo da educação básica como auxílio ao pensamento computacional (SOUZA; FALCÃO; MELLO, 2021).
- *Lógica.* Decisões são tomadas a partir de pensamento sistemático e racional. Mas, porquê isso é um motivo? Trabalhar com a lógica para determinar as ações envolve um domínio matemático, ou está certo, ou está errado. Logo, não existe espaço para o talvez fazendo com que você tenha o controle das ações (ALMEIDA; VALENTE, 2019).
- *Softwares.* Você alguma vez se perguntou como alguns softwares são feitos ao invés de simplesmente utilizá-los? Essa pergunta vai principalmente para quem já desmontou algum brinquedo na vida. Se buscar entender o funcionamento de algo faz sentido

para você, então você possui a tal curiosidade. Embora, não iremos fazer isso durante o curso, desmontar coisas, daremos um pontapé inicial para começar a compreender os softwares e tecnologias que utilizamos diariamente.

- *Você já programa.* Como assim? Quantos problemas você enfrentou ou resolveu na vida? Creio que muitos! Programar é ensinar uma máquina a resolver problemas. Certo! Sei que muitos dos seus problemas não podem ser resolvidos por um computador, mas muitos podem ser resolvidos com auxílio de algoritmos que fazem parte da vida tecnológica, a qual estamos a cada dia mais dependentes. Os algoritmos nos mostram como podemos traçar mecanismos para resolver nossos problemas reais.

Um dos maiores medos e resistências para iniciar a estudar programação é a seguinte pergunta: *é preciso saber inglês?* Não, não é preciso saber inglês. Eu mesmo não sabia nada de inglês, aqui não estou considerando as disciplinas de inglês do ensino fundamental e médio, que era o básico do ensino concentrado no *to be*, logo para o que iria ver na programação considero não ter visto nada de forma profunda. No entanto, para aprender algumas linguagens de programação, seja preciso conhecer algumas palavras em inglês, tais palavras são conhecidas como *palavras reservadas*. Nos próximos capítulos falaremos mais delas. Agora falando para os amantes da língua inglesa, o mundo anseia por programadores, logo se você domina a língua e sabe programar existem vagas no mercado de trabalho à sua espera (MORZE; MASHKINA; BOIKO, 2022).

Então, saiba que programar é uma tarefa que exige tempo e dedicação para ser corretamente aprendida. Não bastando somente estudar e fazer os exemplos, mas principalmente em deixar a mente se acostumar com uma nova forma de pensar (MENEZES, 2010). Para iniciarmos nossa jornada veremos como usar a *lógica*.

1.2 Lógica

Engraçado como essa palavra na boca de uma pessoa que faz programação significa algo extremamente básico. Isso envolve o pensamento computacional falado anteriormente.

A lógica foi utilizada por pensadores, filósofos, como uma forma de formular pensamentos, ideias e constituir teoremas e conceitos. Neste ponto de vista, a lógica trata da correção do pensamento. Como filosofia, ela procura saber por que pensamos assim e não de outro jeito. Como arte ou técnica, ela nos ensina a usar corretamente as leis do pensamento (VIEIRA et al., 2022).

Irei fundamentar a lógica aqui, com base no seguinte argumento: premissas nos levam a conclusões. Pensemos nas premissas como verdades. Logo, com a organização do conhecimento prévio (as premissas) podemos organizar o pensamento e concluir se algo é verdadeiro ou falso. Eu presumo que, talvez, você não tenha entendido, então vamos a um exemplo para ficar mais fácil o entendimento.

■ **Exemplo 1.1** Temos as seguintes *premissas*:

1. O livro está no armário.
2. O armário está fechado.

Podemos tirar as seguintes *conclusões*:

1. Para pegar o livro eu terei que abrir o armário.
2. O armário contém pelo menos um livro.

■

Observe que todas as conclusões que adquirimos são baseadas unicamente nas premissas que nos foram dadas. Logo, eu não posso "inventar" conclusões desconsiderando a lógica dada pelas premissas. Por exemplo, a seguinte conclusão: o armário contém muitos livros. Nada pode ser afirmado, pois nenhuma premissa nos disse a quantidade de livros que estão

dentro do armário, o que sabemos é que pelo menos um livro ele possui (primeira premissa). Entendeu a ideia? Vamos ver mais dois exemplos:

■ **Exemplo 1.2** Temos as seguintes *premissas*:

1. Eu sou mais velho que João.
2. João é mais velho que José.

Podemos tirar as seguintes *conclusões*:

1. Eu sou mais velho que José.
2. José é mais novo que eu.

■

■ **Exemplo 1.3** Temos as seguintes *premissas*:

1. Todos os filhos de José são mais altos do que Maria.
2. Antônio é filho de José.

Podemos tirar as seguintes *conclusões*:

1. Antônio é mais alto que Maria.
2. Maria é mais baixa que Antônio.

■

No Exemplo 1.3, se eu dissesse a seguinte conclusão: José possui pelo menos uma filha. Estaria a conclusão correta ou errada? Se sua resposta foi errada, você acertou. Pois, em nenhuma premissa temos que Maria é filha de José.

Observa-se que essas premissas e conclusões textuais podem conter "pegadinhas", logo, devemos ter muita atenção. Isso me lembra muito do ensino médio, onde essa parte de lógica eu não me dava tão bem. Por este motivo gosto de programação misturada com matemática, lugar que a lógica é mais palpável. Vejamos um exemplo disso:

■ **Exemplo 1.4** Temos as seguintes *premissas*:

1. $x \geq 5$.
2. $y \leq 10$.
3. $z > x$ e $z \leq y$.

Podemos tirar as seguintes *conclusões*:

1. $z \neq 5$
2. $x < 10$

■

No Exemplo 1.4, embora sejam expressões matemáticas, elas indicam o que indicam e ponto final. Mas, talvez você se pergunte como chegamos a essas conclusões, então vamos lá. Inicialmente, tenha ciência que não sabemos os verdadeiros valores de x, de y ou de z. A primeira conclusão é obtida a partir da premissa 1. e 3., x pode ser maior ou igual a 5, mas z é somente maior que x, logo o único valor que temos certeza que z não pode assumir seria 5. Certo? A segunda conclusão obtemos a partir das três premissas, mas nos baseamos principalmente na 3., nessa premissa é dito que $z > x$ e $z \leq y$, observe que o z não pode ser maior que y e que z é maior que x, logo podemos concluir que z é sempre superior a x, logo o x terá um limite superior, ou seja, um valor que é menor que y (assumindo que z possa ser 10). Creio que foi um pouco complexo processar esse exemplo, certo? Então, como exercício, tente verificar o texto deste parágrafo com a Figura 1.1.

Na Figura 1.1, observam-se três exemplos de possíveis valores que essas variáveis, x, y e z, entre infinitas possibilidades.

Concluindo, temos que as conclusões são obtidas a partir de fatos (premissas) e também a partir de outras conclusões. Isso tudo é essencial para trabalharmos com programação e é utilizado para desenvolver pesquisa científica.

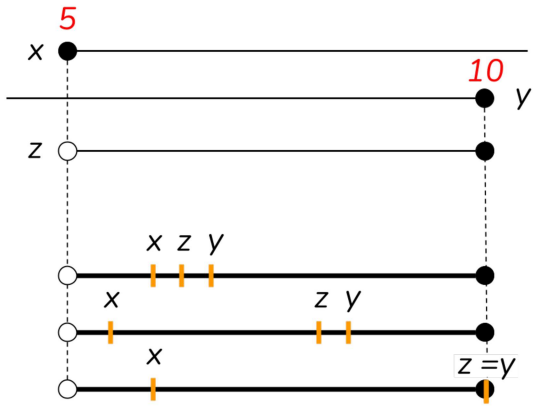


Figura 1.1: Possíveis valores das variáveis do exemplo 1.4 na reta. O símbolo de círculo preenchido preto indica que o valor sobre o símbolo pode ser assumido pela variável (ex., x pode ser igual a 5). O símbolo de círculo vazado indica que a variável não pode assumir o valor (ex., $z \neq 5$). Os três últimos segmentos de reta indicam possíveis localizações para cada variável.

1.3 Exercícios

Exercício 1.1 Se em um semáforo a luz vermelha acesa é para o motorista parar e a luz verde acesa é para seguir, estando você na faixa de pedestre para atravessar a rua. Quais das alternativas são verdadeiras (considerando que você não quer ser atropelado(a)):

a) Pode-se atravessar a rua quando a luz vermelha estiver acesa.
b) Pode-se atravessar a rua quando a luz verde estiver acesa.
c) O semáforo possui duas luzes.
d) O semáforo possui pelo menos duas luzes.

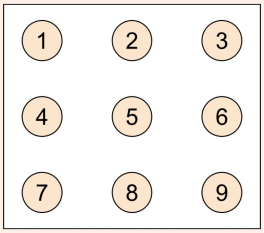
■

Exercício 1.2 Patos são animais. Patos têm duas patas. Qual a conclusão:

a) Todo animal possui duas patas.
b) Patos não possuem quatro patas.
c) Patos tem bico.

■

Exercício 1.3 Desafio dos nove pontos o objetivo é traçar quatro linhas retas passando por todos os nove pontos, sem tirar o lápis/caneta do papel. Para facilitar o raciocínio e a resolução, marque os nove pontos em uma folha de papel e tente resolver.



■

2. Algoritmo e suas Representações

Neste capítulo você aprenderá a:

- Identificar um algoritmo no dia a dia;
- Diferenciar entre os tipos de representação de algoritmos;
- Aplicar algoritmos na resolução de problemas.

2.1 Algoritmo

Na minha interpretação, o algoritmo pode ser entendido como um molde para o que virá a se tornar um programa, ou seja, é um elemento que possui todos os itens necessários para desenvolvermos nosso programa. Certo, talvez esta palavra seja nova, algoritmo, e o que eu disse possa não fazer muito sentido para você, então vamos à uma analogia.

Pare por alguns minutos e reflita sobre o que você fez em um determinado momento do dia, como, por exemplo: do acordar até tomar café da manhã. No meu caso, posso listar as que ocorreram as seguintes ações nesta ordem:

1. Levantei da cama;
2. Coloquei comida para as gatas;
3. Levei os cachorros para o quintal;
4. Arrumei a cama;
5. Levei as gatas para o quintal;
6. Preparei o café;
7. Tomei café.

Observe que os passos realizados estão em uma linguagem que não expressa detalhes do que foi feito, em cada uma dessas ações, é o que chamamos de linguagem de *alto-nível*. Embora, eu não tenha descrito detalhes do que houve em cada ação, conseguimos identificar uma ordem e o que foi realizado. Esse passo a passo, cronológico, é um exemplo de algoritmo.

Outro fato interessante de um algoritmo é que ele determina uma rotina. A rotina é algo que pode ser replicada, como a nossa rotina matinal. Essa característica do algoritmo é essencial para desenvolvermos aplicações escaláveis (que evoluem), pois não é preciso

começar do zero; podemos reaproveitar rotinas e sub-rotinas, caso necessitemos, realizadas por nós mesmos ou por terceiros. Na pesquisa científica em computação, os autores devem mostrar em seus artigos, quando envolver a produção de um programa (implementação), os algoritmos utilizados. Tendo como objetivo permitir com que outros pesquisadores possam replicar os seus resultados. Isso permite avanço científico.

Ao unir minha interpretação e o exemplo vamos para um conceito de algoritmo mais formal. O algoritmo é uma especificação de uma sequência ordenada de instruções, finitas e não-ambíguas (sem redundância), que deve ser seguida para a solução de um determinado problema, garantindo a sua repetibilidade (JUNIOR, 2009). A partir dessa definição você consegue visualizar um exemplo de algoritmo no seu dia a dia? Vou citar um bem simples: receitas culinárias. Elas mostram passos (ordenados) para se produzir como produto final: um prato delicioso, ou pelo menos é o que esperamos.

Creio que temos agora uma noção melhor do que seja um algoritmo, mas você deve querer saber como o utilizaremos para programação. Na Seção 1.1, falei que temos que deixar a mente se acostumar com uma nova forma de pensar. O algoritmo é essa nova forma de pensar. Suponha que você quer beber um copo com água, como você pensaria em um algoritmo para isso? Vamos ver:

1. Primeiro, terá que pegar um copo;
2. Pegar um recipiente (garrafa, torneira, bacia,...) que contenha a água que quer beber;
3. Transferir a água do recipiente ao copo até uma quantidade suficiente;
4. Por fim, beber a água contida no copo.

Ufa! Cansou só de ler, né? Um algoritmo deve ser algo "mastigado". Se eu falasse "eu vou pegar copo d'agua", o recado estava dado, mas para um computador não é bem assim que funciona. Esses exemplos simples e ilustram a ideia de um algoritmo, agora vamos partir para problemas mais palpáveis.

Lembra das aulas de matemática, nas quais você tinha que calcular a raiz de uma equação de primeiro grau? Se você utilizasse algoritmos para resolver este problema sua vida seria bem mais prática, tenho certeza. Vamos ver na próxima seção esse e outros exemplos de como aplicar a representação de algoritmos: para o cálculo de um desconto e para o cálculo da média entre dois valores.

2.2 Representações de Algoritmos

Entre as representações mais comuns de um algoritmo destacaremos nesta seção três: **descrição narrativa**, **fluxograma** e **pseudocódigo**. A principal diferença entre elas está no maior ou menor nível de detalhamento (grau de abstração). Quanto mais detalhes teremos de como implementar esse algoritmo, ou seja, criar um programa a partir da representação mais alto será o grau de abstração. Talvez você possa perguntar: existe uma representação que seja a melhor? A resposta para essa pergunta é: depende!

Cada programador pode optar por um ou outra representação para estruturar o seu algoritmo. Irei definir três problemas para podermos observar cada representação e, assim, conseguirmos analisar e comparar essas representações.

Problema 2.2.1 — Calcular o desconto de um produto. Esse problema é recorrente em muitos casos. Quem já fez aquela pergunta, tem desconto? Levanta a mão! (eu). Vamos criar uma fórmula para isso. Se temos um produto que custa um `valor` em reais e queremos calcular uma `porcentagem` deste `valor` (`desconto`) devemos realizar o seguinte cálculo:

$$\texttt{desconto} = \texttt{valor} \times \frac{\texttt{porcentagem}}{100} \tag{2.1}$$

onde *desconto* é o valor que buscamos.

Por exemplo, qual o desconto de 25% em um produto que custa 150 reais? O resultado seria: $\texttt{desconto} = 150 \times \frac{25}{100}$, logo $\texttt{desconto} = 37{,}50$ reais.

Problema 2.2.2 — Calcular a média entre dois valores. Todo/a estudante provavelmente deve ter enfrentado este problema, a tal média bimestral. O cálculo da média aritmética entre dois valores consiste em somar os dois valores e dividir por dois (2). Formalizando, a `media`, entre dois valores $\texttt{n}_1$ e $\texttt{n}_2$ pode ser calculada com a seguinte fórmula:

$$\texttt{media} = \frac{\texttt{valor}_1 + \texttt{valor}_2}{2} \tag{2.2}$$

Por exemplo, se $\texttt{n}_1 = 8$ e $\texttt{n}_2 = 10$, $\texttt{media} = \frac{8+10}{2}$, fazendo os cálculos a $\texttt{media} = 9$; logo, a média entre esses dois valores, 8 e 10, é 9. Desconsiderem o erro ortográfico de *média* na fórmula, depois saberá o motivo.

Problema 2.2.3 — Calcular a raiz de uma equação de primeiro grau. Toda equação de primeiro grau é uma função que possui uma única incógnita elevada a 1 (usualmente não se coloca o 1). Podemos representar essa função do seguinte modo: $f(\texttt{x}) = \texttt{ax+b}$. Para encontrar a raiz desta função devemos igualá-la a zero e isolar o `x`:

$$\texttt{ax+b} = 0 \tag{2.3}$$

$$\texttt{x} = \frac{-\texttt{b}}{\texttt{a}} \tag{2.4}$$

onde, `x` é o valor que representa a raiz da equação. Observe que o valor de `a` deve diferir de zero ($\texttt{a} \neq 0$), senão teremos uma indeterminação.

Por exemplo, na função $f(\texttt{x}) = 5\texttt{x} - 10$, para encontrar a função faremos $f(\texttt{x}) = 0$, que é a mesma coisa que $5\texttt{x} - 10 = 0$, e isolando `x` temos: $\texttt{x} = 2$. Logo, 2 é a raiz dessa equação.

Com base no entendimento desses problemas vamos agora conhecer cada representação. É imprescindível que você compreenda cada problema. Por isso, leia com calma cada um dos problemas novamente, caso precise, para então prosseguir com as representações.

Este capítulo teve como grande contribuição os trabalhos de (ALGORITMO..., 2004; VASCONCELLOS; TAMARIZ; BATISTA, 2019).

2.2.1 Descrição Narrativa

Na descrição narrativa os algoritmos são expressos diretamente em linguagem natural. Ou seja, uma sequência de passos é descrita em nossa língua. É só lembrar dos nossos exemplos na Seção 2.1, rotina e beber água, onde utilizamos esse tipo de descrição.

Esta representação não é tão boa por permitir várias interpretações, dificultando transcrição para um programa. Por exemplo: se eu falar "rebolar no mato" um piauiense, ou cearense, possivelmente saberia o significado: *jogar no lixo*. Porém, pode acontecer que alguém de fora destas regiões possa não compreender bem e pensar que alguém pessoa irá

dançar, rebolar, em uma área com mato.

Agora vamos aos exemplos ordenados conforme os Problemas 2.2.1, 2.2.2 e 2.2.3.

■ **Exemplo 2.1** Problema 2.2.1 (Desconto):

1. Obter `valor`.
2. Obter `porcentagem` (de 0 a 100).
3. Calcular o `desconto`.

■

■ **Exemplo 2.2** Problema 2.2.2 (Média Aritmética):

1. Obter o valor da primeira nota, $\mathtt{n}_1$.
2. Obter o valor da segunda nota, $\mathtt{n}_2$.
3. Calcular a média (`media`).

■

■ **Exemplo 2.3** Problema 2.2.3 (Raiz da Equação de Primeiro Grau):

1. Obter o valor de `a`.
2. Obter o valor de `b`.
3. Se `a` é igual a zero, dizer que não existe raiz para a equação.
4. Senão, a raiz da equação será $-\frac{\mathtt{b}}{\mathtt{a}}$.

■

Bem simples, né!? Observa-se que com a descrição narrativa podemos facilmente resolver nossos problemas. Porém, em algumas situações não foram detalhados alguns procedimentos, como no cálculo do desconto (Exemplo 2.1).

2.2.2 Fluxograma

É uma representação gráfica em que cada forma descreve uma ação distinta (instruções, comandos, direções, etc). Por se tratar de um sistema de representação visual simplificado é mais precisa que a descrição narrativa, pois minimiza a ambiguidade. É como uma placa de trânsito, você sabe seu significado de imediato ao vê-la (embora muitos confundam as placas de proibido estacionar e de proibido parar e estacionar). Aqui, começaremos a ver a silhueta da estrutura de um programa, com *comandos* de entrada, saída, e atribuição de dados, além de comandos de decisão (condicionais) (Figura 2.1). Além disso, temos um símbolo que representará a direção do fluxo, uma seta.

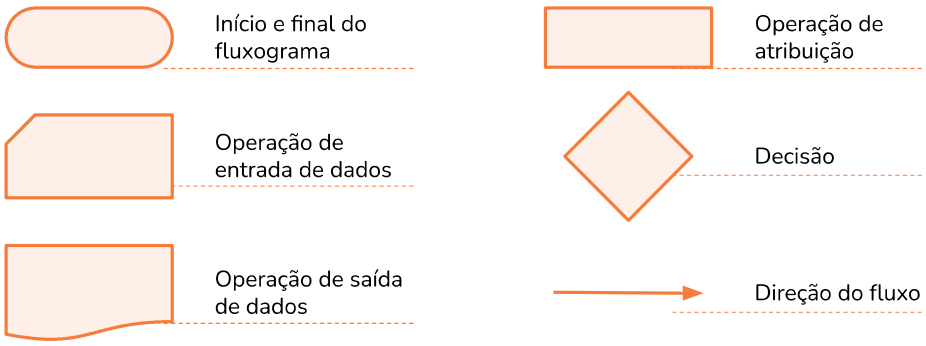


Figura 2.1: Símbolos presentes na representação de fluxograma e seus significados.

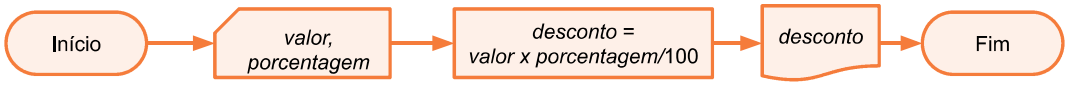
Os símbolos que usaremos estão dispostos na Figura 2.1. Cada símbolo tem seu significado próprio. Em um fluxograma sempre existirá um início e pelo menos um fim. Posso destacar que as operações que serão recorrentemente utilizadas são: operação de

entrada de dados, onde o usuário irá informar um valor e o mesmo será atribuído a uma variável; operação de saída de dados onde algo será impresso (exibido na tela); e operação de atribuição, onde um valor será atribuído a uma variável.

O símbolo de decisão, losango, indica que o fluxo será dividido em duas partes, caso seja verdadeiro seguirá uma direção, caso contrário seguirá outra direção.

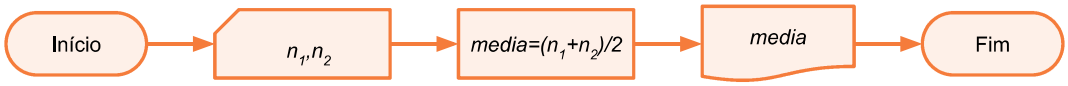
Vejamos cada um dos problemas agora utilizando fluxogramas.

■ **Exemplo 2.4** Problema 2.2.1 (Desconto):



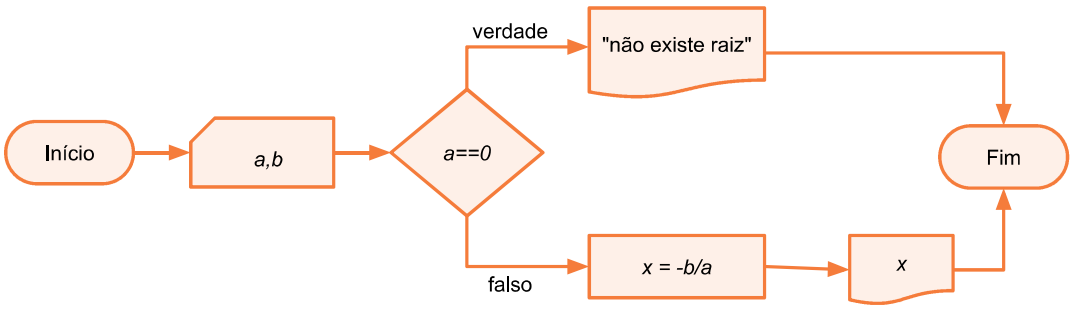
Observe que podemos obter mais de um valor utilizando uma vírgula, ao invés de usar duas formas para entrada de dados. Após o cálculo do desconto ele será impresso pelo algoritmo, caso não tenha uma operação de impressão nunca saberemos o que foi realizado pelo algoritmo. ■

■ **Exemplo 2.5** Problema 2.2.2 (Média Aritmética):



■

■ **Exemplo 2.6** Problema 2.2.3 (Raiz da Equação de Primeiro Grau):



Aqui temos um símbolo de decisão. A representação de igualdade é feita com dois símbolos de igual juntos (==). Caso seja verdade, ou seja, `a` é igual a zero, a raiz não existe e seguimos a direção da impressão do texto, delimitado com aspas duplas, informando isso. ■

Observa-se que o fluxo descreve o passo a passo do algoritmo. Essa forma de representação é excelente para visualizar como o algoritmo deve funcionar.

2.2.3 Pseudocódigo

Esta representação é rica em detalhes, tanto na estrutura como na definição dos tipos das variáveis usadas no algoritmo. Aqui, utiliza-se, também, a linguagem natural para estruturar a representação em conjunto com palavras-chave que serão os *comandos*.

Entre os comandos destaco os seguintes:

- **Entrada**: definição das variáveis de entrada;
- **Saída**: definição da saída do algoritmo;
- **leia**: indica uma operação de entrada de dados;
- **imprime**: indica uma operação de saída de dados;
- $\leftarrow$: significa uma atribuição;
- **se**: verifica se uma condição é *verdadeira*, caso seja é executado o que tiver dentro deste escopo;
- **senão**: define um escopo que será executado caso a condição do **se** tiver sido *falsa*. Um **senão** só pode ser definido associado a um **se**.

■ **Exemplo 2.7** Problema 2.2.1 (Porcentagem): ■

Algoritmo 2.1: Calcula Desconto

Entrada: *valor*, *porcentagem*
Saída: *desconto*
1 $desconto \leftarrow valor \times \frac{porcentagem}{100}$
2 **retorna** *desconto*

No Exemplo 2.7, defini na **Entrada** quais os valores que precisamos possuir para que o algoritmo funcione. E em **Saída**, defini o que o algoritmo irá retornar, para ser impresso na tela, por exemplo. Esses comandos poderiam ser substituídos por **leia** e **imprima**, respectivamente.

■ **Exemplo 2.8** Problema 2.2.2 (Média Aritmética): ■

Algoritmo 2.2: Calcula Média de Dois Valores

1 **leia** $nota1$
2 **leia** $nota2$
3 $media \leftarrow \frac{nota1 + nota2}{2}$
4 **imprime** *media*

No Exemplo 2.8, o nome da variável sem o acento (`media`) não foi um descuido. Vamos nos acostumar a não acentuar as nossas variáveis, pois as linguagens de programação não entendem alguns símbolos, veremos isso com detalhes no próximo capítulo.

■ **Exemplo 2.9** Problema 2.2.3 (Raiz da Equação de Primeiro Grau):

Algoritmo 2.3: Calcula Raiz da Equação de Primeiro Grau

Entrada: a, b
Saída: x
1 **se** $(a == 0)$ **então**
2 | **imprime** "não existe raiz"
3 | **retorna**
4 **senão**
5 | $x \leftarrow -\frac{b}{a}$
6 | **retorna** x

■

No Exemplo 2.9, o símbolo de == é um operador lógico de igualdade, ele indica se a igualdade é verdadeira ou falsa. Por exemplo, $5 == 2$ retorna *falso*, e $7 == 7$ retorna *verdadeiro*. No caso, na linha 1 está sendo verificado se a é igual a 0. Se for igual a zero será executado o escopo do **se** (linhas 2 e 3), pois a condição é *verdadeira*; caso contrário será executado o escopo do **senão** (linhas 5 e 6). Podemos encarar a condição $(a == 0)$ como uma premissa (algo verdadeiro) para que o escopo do **se** seja executado.

Oriento que tente realizar mais o uso do pseudocódigo para se familiarizar com a estrutura dos programas que faremos utilizando a linguagem de programação *javascript*.

2.3 Resumo

Podemos agora observar a Tabela 2.1 com olhar mais criterioso enxergando as vantagens e desvantagens de cada abordagem. Em geral, os programadores utilizam mais as representações de fluxograma e pseudocódigo para construção de seus algoritmos por direcionar melhor a construção do programa, principalmente o pseudocódigo.

Representação	Grau de Abstração	Vantagens	Desvantagens
Descrição Narrativa	Baixo	A língua natural é conhecida, não é preciso entender novas palavras e linguagens	A língua natural por si só abre margem para ambiguidades, dificultando a construção do programa a partir do algoritmo
Fluxograma	Médio	Conhecendo os símbolos (gráficos) é mais fácil entender o fluxo do algoritmo	Sendo uma versão simplificada do algoritmo o fluxo pode não possuir detalhes suficientes para construção do programa
Pseudocódigo	Alto	Representação clara mesmo sem conhecer as especificações de linguagem de programação	As regras do pseudocódigo devem ser conhecidas

Tabela 2.1: Resumo das representações de um algoritmo com vantagens e desvantagens.

2.4 Exercícios

Exercício 2.1 Entre os exemplos abaixo, NÃO pode ser considerado um algoritmo:

a) Guia de montagem de um guarda-roupas.
b) Manual de instruções de uso do celular.
c) Receita de sorvete.
d) Cardápio da pizzaria.

■

Exercício 2.2 A afirmação "O algoritmo é uma sequência de passos lógicos e finitos e não-ambíguos que permitem solucionar problemas" é:

a) Verdadeira.

b) Falsa.

■

Exercício 2.3 A afirmação "É um consenso entre os programadores que a melhor forma de representação de um algoritmo é a descrição narrativa"é:

a) Verdadeira.
b) Falsa.

■

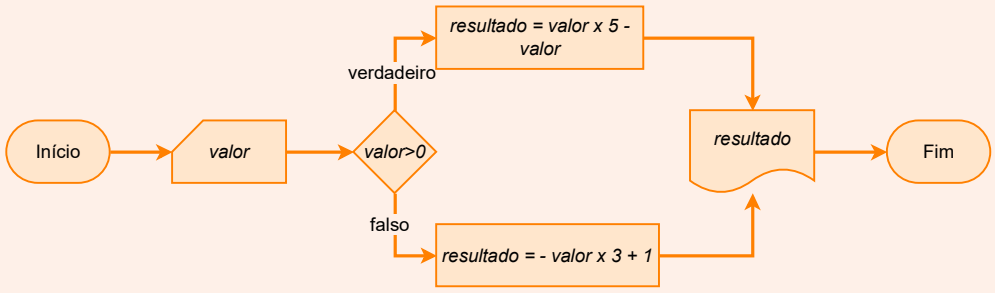
Exercício 2.4 Faça o algoritmo que verifique se uma pessoa é maior de idade conforme a idade informada. Utilize qualquer uma das representações para desenvolver seu algoritmo. ■

Exercício 2.5 Faça um algoritmo que verifica se um número informado é positivo ou negativo. Utilize qualquer uma das representações para desenvolver seu algoritmo. ■

Exercício 2.6 Faça um algoritmo que verifica se a média aritmética de três notas aprova ou reprova um(a) aluno(a). O(A) aluno(a) é considerado(a) aprovado(a) se sua média for maior ou igual a 7. Utilize qualquer uma das representações para desenvolver seu algoritmo. ■

Exercício 2.7 O ***jokenpo*** é um jogo que é jogado entre duas pessoas colocando uma configuração de mão: Pedra (✊), Papel (✋) e Tesoura (✌). As regras são as seguintes: ✌ ganha de ✋ e perde para ✊; ✊ ganha de ✌ e perde para ✋; e ✋ ganha de ✊ e perde para ✌. Caso os jogadores coloquem a mesma configuração de mão é empate. Caso um dos jogadores ganhe você deve informar qual jogador ganhou. Utilize qualquer uma das representações para desenvolver seu algoritmo. ■

Exercício 2.8 Utilizando o fluxograma a seguir responda cada um dos itens.



a) Se o *valor* de entrada for 5, o que irá ser impresso?
b) Se o *valor* de entrada for -2, o que irá ser impresso?
c) Se o *valor* de entrada for 0 (zero), o que irá ser impresso?
d) Faça um pseudocódigo a partir deste fluxograma.

■

II Conhecendo os Comandos Básicos

3. Tipos de Dados e Variáveis

Neste capítulo você aprenderá a:

- Identificar os tipos de dados básicos utilizados na programação;
- Utilizar o Replit para praticar a programação em *javascript*;
- Criar variáveis e constantes em *javascript*.

3.1 Tipos de Dados

Parabéns por chegar até aqui, a partir de agora todos os conceitos da Parte I serão estruturados para uma linguagem de programação. Como linguagem optei por usar o *javascript*, por ser mais acessível, visto que todo navegador de internet o possui nativamente. Nossa jornada a partir de agora será conhecer os tipos de dados e de variáveis. Vimos na Seção 2.2 que as variáveis são essenciais para o funcionamento de todos os algoritmos.

Podemos destacar dois tipos de informações que toda linguagem de programação possui: instruções (comandos) e dados.

- Instruções: determinam como os dados serão tratados, cada computador pode processar de uma forma diferente. Exemplos: comandos de entrada, comandos de saída, comandos condicionais (*se-senão*), entre outros.
- Dados: representam uma porção de informações que serão processadas, manipuladas, pelo computador. Exemplo: os valores de a, b e x do Exemplo 2.9.

Você deve ter percebido que todas as variáveis tratadas nos pseudocódigos (Seção 2.2.3) estavam em um estilo de texto diferente. Essas variáveis são representadas por um tipo de dado. Faço uma restrição neste material introdutório a três dados básicos: numérico, literal e lógico (Figura 3.1). Porém, existem outros tipos de dados que são específicos de cada linguagem de programação; no caso do *javascript* existem os tipos: `function` (função), `object` (objeto), `undefined` (indefinido) e `null` (nulo).

Na Figura 3.1, temos três tipos de dados básicos:

1. Literal (`string`): representam uma sequência de caracteres, que podem ser formados por: letras, dígitos e símbolos especiais, delimitados por aspas duplas, aspas simples

ou crase (no *javascript*). Exemplos: `"A","Hoje irei programar", 'Como está?',` `Bem!;)`. Observe que o texto inicia por um desses símbolos e finaliza com o mesmo que iniciou. Não se pode misturar a representação. Caso opte por usar aspas duplas inicie e finalize o literal com aspas duplas.

> A representação mais utilizada, na maioria das linguagens de programação, é a de aspas duplas.

2. Numérico (`number`): correspondem aos números, são divididos em reais e inteiros. Os números reais são números que possuem uma parte fracionada (indicada por uma vírgula), por exemplo: 3,54. Inteiro é um número que não possui vírgula, por exemplo: 10.

> A vírgula nos números reais serão substituídos em nossos programas por ponto. Logo, se você quiser utilizar o valor 3,54, então no seu programa você deverá escrever `3.54`, certo!?

3. Lógico (`boolean`): São usados para representar dois únicos valores possíveis, verdadeiro (`true`) ou falso (`false`).

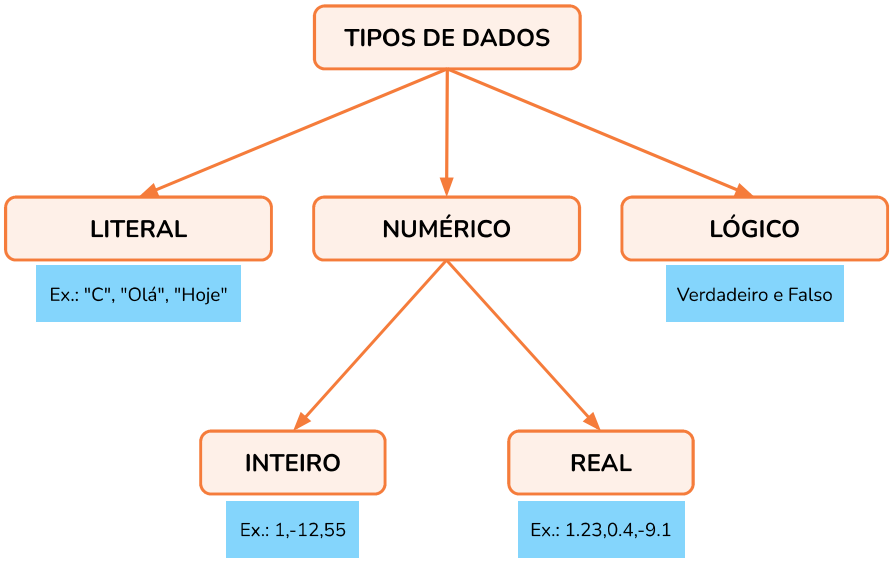


Figura 3.1: Tipos de dados básicos com exemplos.

3.2 Tipos de Dados no Javascript

Agora vamos ver esses três tipos de dados básicos no *javascript* e como identificá-los. Inicialmente, irei mostrar como você pode reproduzir todos os exemplos no seu computador. Por praticidade utilizei o software Replit[1] (site) para hospedar todos os códigos. Criei, também, uma lista com tutoriais[2] em forma de vídeo para que consiga executar os primeiros passos neste software. A linguagem que deverá ser selecionada no Replit para realização das implementações será o **nodejs**, uma linguagem baseada em *javascript* (TILKOV; VINOSKI, 2010).

[1] <https://replit.com/>
[2] <https://bit.ly/tutoriais-replit>

Na Figura 3.2, destaco as principais áreas que iremos trabalhar no Replit. Seguem algumas informações adicionais relacionadas a cada indicador da figura:

A) É o nome do projeto que você define ao criar um replit. Pode ser alterado sempre que quiser.

B) É o botão utilizado para executar o programa. Caso deseje que o programa em execução pare basta clicar nele novamente.

C) Nesta área ficarão listados os arquivos do seu projeto. Em nossos exemplos e práticas usaremos, somente, um arquivo por projeto.

D) O nome do arquivo principal do nosso projeto é o `index.js` (por padrão). O `.js` no final do arquivo determina o tipo de arquivo: *javascript*. Você não precisará alterar o nome desse arquivo em nossos exemplos e práticas.

E) Nesta área colocaremos o código-fonte do programa, que será executado ao apertar no botão verde (indicador B).

F) Nesta área, denominada `console`, você verá a saída de dados do seu programa. E, também, é a área de entrada de dados, onde você irá inserir os dados de entrada que seu programa irá processar.

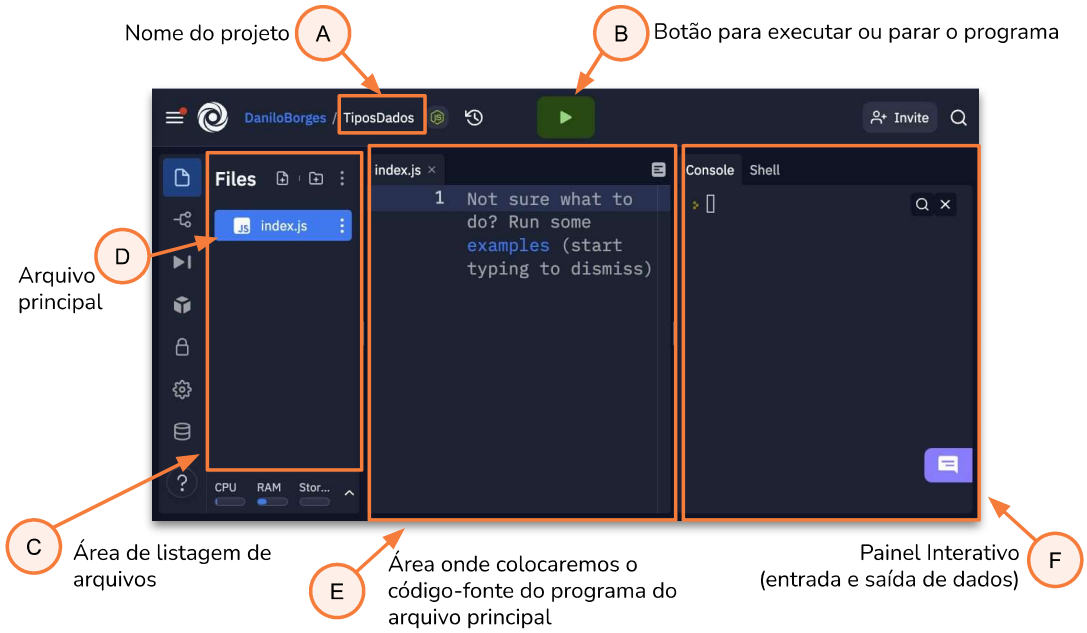


Figura 3.2: Tela de um projeto de programação no Replit. Em destaque as principais áreas do projeto utilizadas nos exercícios e práticas.

Vamos focar na área do painel interativo (Figura 3.2F), o `console`, para observar os tipos de dados. Cada linha será um comando. O comando finaliza com um *enter* do teclado. Vamos ver um exemplo? Trabalharemos com o comando `typeof()`. Este comando mostra o tipo de dado considerado no *javascript*. Para utilizá-lo colocaremos entre parenteses um dado. Como exemplo usaremos os seguintes comandos:

```
1 typeof(34);
2 typeof(37.5);
3 typeof('A');
4 typeof(`Hoje`);
5 typeof("Tudo bem?");
6 typeof(5>10);
```

Para melhor organizarmos nosso código, encerraremos este comando `typeof()` com um ponto e vírgula (;).

Verifique se conseguiu reproduzir os mesmos resultados no Replit, como mostra a Figura 3.3. Esses serão os tipos de dados que iremos utilizar em nossas variáveis.

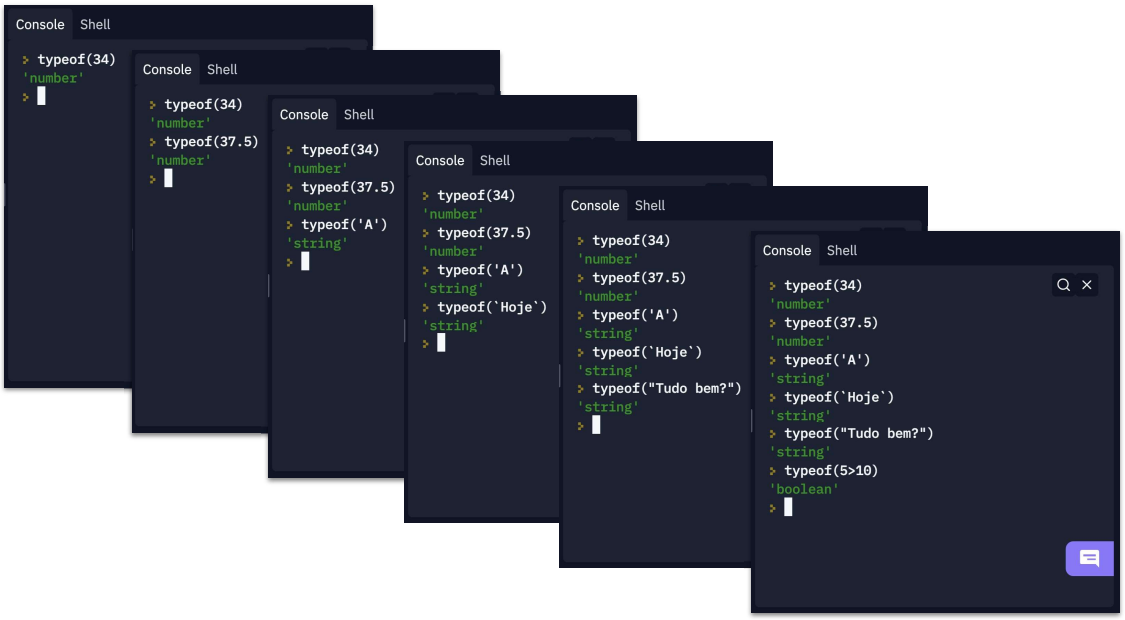


Figura 3.3: Execução de código de verificação de tipo passo a passo no Replit.

3.3 Variáveis

Como o próprio nome diz, variável é algo que pode variar, ou seja, pode ter seu valor alterado. No contexto da programação, as variáveis são identificadores que irão armazenar um tipo de dado recebido em uma atribuição. Como no Exemplo 2.1 onde `valor` armazena o preço de um produto. Pensando neste mesmo exemplo, qual tipo de dado você acha que seria armazenado na variável `valor` no *javascript*? Você está certo se tiver respondido `number`.

3.4 Variáveis no Javascript

Para definir uma variável você precisará utilizar uma palavra-chave: `var` seguindo de um identificador. Utilizando o `valor` do Exemplo 2.1, nosso identificador seria o mesmo nome, então ficaria assim:

```
1 var valor;
```

Para melhor organizarmos nosso código, encerraremos este comando com um ponto e vírgula (;).

Neste exemplo a variável `valor` foi criada, mas não possui nenhum valor atribuído. Para atribuir um valor para variável podemos fazer do seguinte modo:

```
1 var valor = 150;
```

Neste caso o `valor` é criado e inicializado com 150.

Outro modo de fazer que produz o mesmo efeito é este:

```
1 var valor;
2 valor = 150;
```

Nesse modo a variável é criada (linha 1) e é posteriormente atribuído um valor a ela (linha 2).

O tipo de dado armazenado na variável depende do dado. No caso da variável `valor` seria um `number`, porém, se fosse realizado o seguinte comando:

```
1 var valor = "Hoje";
```

`valor` seria uma `string`. Entendeu!? A última atribuição no `valor` é que definirá seu tipo de dado. Para ficar mais claro, vejamos este outro exemplo:

```
1 var valor;
2 valor = "Hoje";
3 valor = 150;
4 valor = 5>2;
```

Qual seria o tipo de dado de `valor` e qual seria seu valor? Resposta correta: tipo de dado `boolean` e valor `true`. Ou seja, `valor` sempre irá receber o valor *setado* (atribuído) em última instância.

Esta característica da linguagem *javascript* de atribuir a uma mesma variável tipos de dados diferentes lhe configura como uma linguagem fracamente tipada. Ou seja, após declarada a variável pode ser sobrecarregada (atualizada) com novo tipo de dado.

Agora que entendemos o que é um identificador vamos às regras... É como criar uma senha, onde existem algumas regras como: colocar pelo menos uma letra maiúscula, ter pelo menos um número, no mínimo oito caracteres, etc.; caso não siga essas regras a senha não é criada. Quando falamos das nossas variáveis se não obedecermos as regras um erro aparecerá e nosso programa irá parar a execução no local do erro. Temos, então as seguintes regras para os identificadores de variáveis:

1. Devem ser iniciadas sempre por uma letra ou *underline*(_);
2. Não devem conter caracteres especiais ($,-,#,!,?,...) exceto o *underline*;
3. Não devem conter espaços em branco.

Podem-se observar no Exemplo 3.1 identificadores que se encaixam nas regras definidas e no Exemplo 3.2 identificadores que falharam em alguma das regras.

■ **Exemplo 3.1** Identificadores válidos no *javascript*:

```
1 var valor;
2 var _endereco0;
3 var data_atual;
4 var Casa;
5 var _0_ok;
```

■

■ **Exemplo 3.2** Identificadores inválidos no *javascript*:

```
1 var valorR#;
2 var _endereço0;
3 var dia de hoje;
4 var 01_nome;
```

Os identificadores da linha 1 e da linha 2, ferem a Regra 2. O da linha 3, fere a Regra 3. E o da linha 4 fere a Regra 1. ■

3.5 Variável Constante no Javascript

Existe um tipo de variável que tem uma característica especial, a imutabilidade – qualidade, estado ou condição de imutável. Essas variáveis são declaradas no *javascript* usando a palavra-chave `const` seguido de um identificador. Além das regras de identificadores que a variável deve seguir temos algumas regras adicionais para o `const`:

1. É obrigatória a declaração de uma constante com atribuição de um valor;
2. Não será possível alterar o valor após a criação da variável.

Ou seja, a variável constante garante que seu valor não será alterado de forma alguma, como a própria definição matemática recomenda. Vejamos:

■ **Exemplo 3.3** Variáveis do tipo constante são encontrados no dia a dia como: nome da capital do Piauí, número do π, número de euler e, o seu nome, nome da mãe, seu CPF. ■

Em várias situações é conveniente o uso das constantes. Vamos aprender a criá-las:

```
1 const CAPITAL_PI = "Teresina";
2 const PI = 3.1416;
3 const EULER = 2.7182;
```

No código acima, note que utilizei somente letras maiúsculas para definir cada variável constante. Isso não é uma regra, porém, para melhor legibilidade do código é um bom padrão de desenvolvimento quando você tiver uma variável constante colocá-la em caixa alta, assim, você saberá que não pode alterar o valor da variável.

3.6 Resumo

Vimos neste capítulo três tipos de dados que nossas variáveis podem ter no *javascript*: `string`, `number`, e `boolean`; pudemos verificar o tipo de dado utilizando a função `typeof()`. Também, vimos como podem ser criadas variáveis e constantes no *javascript* e como são as regras de criação de identificadores.

Para praticar execute as seguintes linhas no `console`:

```
1 var valor = "Teresina";
2 typeof(valor)
3 var valor = 3.1416;
4 typeof(valor)
5 valor = true;
6 typeof(valor)
```

Você verá que os resultados de cada `typeof` devem ser, em ordem, `string`, `number` e `boolean`.

3.7 Exercícios

Exercício 3.1 Entre as opções abaixo quais estão relacionadas aos tipos de dados básicos:

a) Inteiro
b) Real
c) Data
d) Literal
e) Imagem
f) Lógico

■

Exercício 3.2 A afirmação "O tipo de dado Lógico pode assumir somente os valores: verdadeiro ou falso"é:

a) Verdadeira.
b) Falsa.

■

Exercício 3.3 Qual item define as informações presentes no tipo de dado literal:

a) Somente letras.
b) Somente números.
c) Somente letras e caracteres especiais.
d) Letras, números e caracteres especiais

■

Exercício 3.4 Sobre variáveis no *javascript* marque TODAS as afirmações verdadeiras:

a) O *javascript* não é uma linguagem fortemente tipada, ou seja, uma variável (`var`) pode mudar seu tipo de dado ao longo do programa.
b) Uma variável constante (`const`) pode ter seu valor alterado após sua criação.
c) Pode ser criada uma variável constante (`const`) sem atribuir nenhum valor inicial.
d) Uma variável (`var`) pode ter seu valor alterado ao longo do programa.

■

Exercício 3.5 Verifique e indique quais dos nomes identificadores abaixo são válidos, nos inválidos, diga o porquê:

a) `voo`
b) `3xct`
c) `_praia`
d) `_apartamento@`
e) `_0_product`

■

Exercício 3.6 Execute o seguinte código em *javascript* e responda cada item.

```
1  var apartamento;
2  apartamento = "Clara"
3  apartamento = "350"
4  apartamento = 350
```

a) Qual o tipo de dado da variável `apartamento`, na linha 2.
b) Qual o tipo de dado da variável `apartamento`, na linha 3.
c) Qual o tipo de dado da variável `apartamento`, na linha 4.

■

4. Operadores e Comandos Básicos

Neste capítulo você aprenderá a:

- Utilizar operadores unários e binários no *javascript*;
- Manipular operadores especiais no *javascript*;
- Utilizar comandos de entrada e saída de dados no *javascript*.

4.1 Operadores

Você já deve ter entendido que programação lida muito com matemática, certo? Se não, então saiba que sim! Operadores são exemplos claros disso. Praticamente todas as operações que vemos nas disciplinas de matemática existem em todas as linguagens de programação, como: soma (`+`), subtração (`-`), divisão (`/`), multiplicação ($\times$). E variações dessas operações. Veremos nesta seção dois tipos de operadores:

- Operadores binários: necessitam de dois dados, ou variáveis, para avaliar e retornar um valor.
- Operadores unários: necessitam de apenas dado, ou variável, para avaliar e retornar um valor.

Talvez ainda não esteja claro o que é o operador unário, embora o tenha utilizado várias vezes. Você alguma vez já fez uma operação como esta: `-(-5)`? De colocar um sinal negativo na frente de um número. Este é um exemplo de operador que só precisa de um operando. Já quando precisarmos de dois operandos nosso operador será binário, exemplo, ao somar `5+9`. Sempre a operação será da esquerda para direita e sempre haverá um retorno para essa operação (Figura 4.1).

O ponto principal desta conversa inicial é que todo operando retorna um valor, caso não fosse dessa forma não teríamos como criar expressões como a do cálculo da raiz da equação de primeiro grau: `x = -b/a`. No cálculo de `x` é utilizado um operador unário, o menos (`-`), e o operador binário, o de divisão (`/`); o resultado dessas duas operações é então atribuído à variável `x` obtendo, assim, o valor desejado do cálculo.

Sem mais delongas vamos ver como trabalhar com os operadores unários e binários

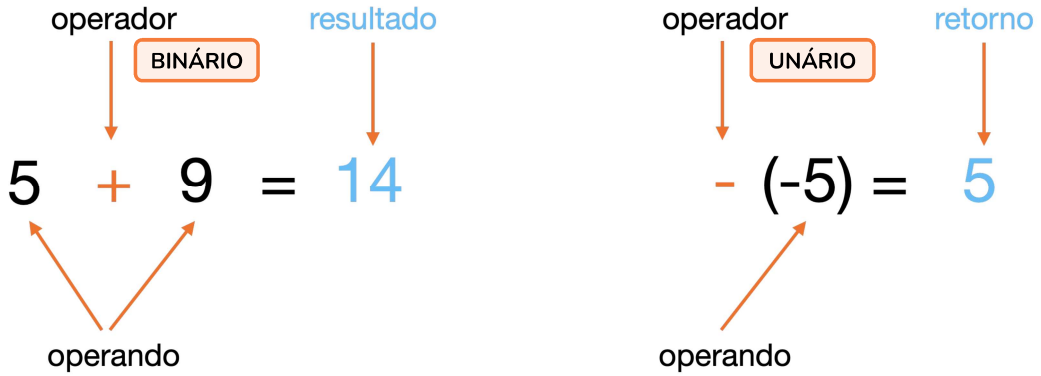


Figura 4.1: Exemplo de operador binário e de operador unário.

no *javascript*. Este capítulo teve como base informações presentes no livro de Flanagan (2004), com algumas atualizações.

4.1.1 Operadores aritméticos

Os operadores aritméticos efetuam operações aritméticas ou outras manipulações numéricas em seus operandos. Os operadores aritméticos básicos são: `**` (potência), `*` (multiplicação), `/` (divisão), `%` (módulo: resto de uma divisão), `+` (adição) e `-` (subtração). A ordem apresentada exibe a ordem de precedência da operação. Caso queira forçar que uma operação seja realizada isoladamente utilize parênteses `()`.

Vejamos alguns exemplos no *javascript*, execute esses comandos no `console`:

```
1  3**2;      //potência, resultado 9
2  3*2;       //multiplicação, resultado 6
3  3/2;       //divisão, resultado 1.5
4  3%2;       //módulo, resultado 1
5  3+2;       //soma, resultado 5
6  3-2;       //subtração, resultado 1
7  2*2**3;    //primeiro potência (2**3 = 8), depois multiplicação (2*8),
       resultado 16
8  (2*2)**3; //primeiro a operação entre parênteses (2*2=4), depois a potê
       ncia (4**3), resultado 64
```

A notação destacada em vermelho representa um *comentário* – uma forma de você falar sobre o código sem que o texto seja processado pelo programa. É como se sua existência não fosse percebida pelo programa. Um comentário no *javascript* é definido com as duas barras juntas (`//`), tudo que estiver à direita delas será desconsiderado pelo programa.

A operação de `%` (módulo) é bastante útil para verificar a divisibilidade entre dois números. Por exemplo: se você quiser saber se um número `x` é par ou ímpar basta verifica o valor de `x%2`. Se o resultado for `0` é porque `x` é par (resto da divisão), se for `1` o número `x` é ímpar.

Caso faça uso dos parênteses lembre-se de ao abir um parênteses, "`(`", não esquecer de fechar, "`)`". Caso isso não aconteça um erro será exibido.

O operador de soma também pode ser utilizado para somar `string`, no caso teremos uma concatenação. Caso seja realizada uma soma de uma `string` e um `number` o resultado será uma `string`. Vejamos alguns exemplos:

```
1  "1"+ "2";                 //resultado: "12"
2  "1" + 2.3;                //resultado: "12.3"
3  "Deu três? " + 1 + 2;     //resultado: "Deu três? 12"
4  1 + 2 + " Deu três?";     //resultado: "3 Deu três?"
```

Observa-se neste exemplo que a ordem em que os operadores ocorrem é sempre da esquerda para direita. Por este motivo na linha 4 houve uma soma (1+2) antes da concatenação.

4.1.2 Operadores aritméticos unários

Os operadores unários modificam o valor de um único operando para produzir um novo valor. Em *javascript*, todos os operadores unários têm precedência alta. Todos operadores aritméticos unários descritos nesta seção: + (mais unário), - (menos unário), ++ (incremento) e -- (decremento); convertem seu único operando em um número, se necessário. Note que os operadores + e - são usados tanto operadores unários como binários.

- +: este operador converte seu operando em um número e retorna esse valor convertido (utilize no caso da `string` conter um número ou no caso de um `boolean` (retorno 0, `false`, ou 1,`true`)).
- -: quando usado como operador unário, ele converte seu operando em um número, se necessário, e depois troca o sinal do resultado.
- ++: este operador soma um (1) ao seu único operando. O operador também converte seu operando em um número antes de proceder com a operação de incremento. O valor de retorno do operador ++ depende de sua posição relativa ao operando.
  Quando usado antes do operando, onde é conhecido como operador de pré-incremento, ele incrementa o operando e retorna o valor incrementado desse operando.
  Quando usado após o operando, onde é conhecido como operador de pós-incremento, ele incrementa seu operando, mas retorna o valor não incrementado desse operando. Pode parecer confuso, mas é bem simples, depois veremos exemplos.
- --: este operador espera um operando. Ele converte o valor do operando em um número, subtrai 1 e atribui o valor decrementado ao operando. Assim como o operador ++, o valor de retorno de -- depende de sua posição relativa ao operando. Quando usado antes do operando, ele decrementa e retorna o valor decrementado. Quando usado após o operando, ele decrementa o operando, mas retorna o valor não decrementado.

Vamos aos exemplos comentados:

```
1  var i = +"1";             //i é um número: 1
2  var i = +"-5";            //i é um número: -5
3  var i = +true;            //i é um número: 1
4  var i = +false;           //i é um número: 0
5  var i = -(5);             //i é um número: -5
6  var i = -true;            //i é um número: -1
7  var i = -false;           //i é um número: -0
8  var i = 1, j = ++i;       //i e j são ambos 2
9  var i = 1, j = i++;       //i é 2 e j é 1
10 var i = 1, j = --i;       //i e j são ambos 0
11 var i = 1, j = i--;       //i é 0 e j é 1
```

Nas linhas 8 a 11 pode-se observar que em uma mesma linha podem ser criadas mais de uma variável com os identificadores separados por vírgula sem colocar `var` novamente.

4.1.3 Operadores de igualdade e desigualdade

Trabalhamos com alguns operadores lógicos como o igual (==) e maior que (>) antes, agora veremos como utilizar no *javascript* outros:

- ==: operador binário que verifica se dois dados são iguais. Retorna `true` caso os dados sejam iguais e `false` caso sejam diferentes.
- !=: operador binário que verifica se dois dados são diferentes. Retorna `true` caso os dados sejam diferentes e `false` caso sejam iguais.

Verificaremos isso com alguns exemplos:

```
1 var i = 50;
2 var j = "Ok";
3 var k = 50;
4 i == j;          //retorna: false
5 i == k;          //retorna: true
6 i != j;          //retorna: true
7 i == j;          //retorna: false
8 i != k;          //retorna: false
```

4.1.4 Operadores de comparação

Os operadores de comparação também retornam valores booleanos, similar aos operadores de igualdade. Iremos trabalhar com os seguintes:

- <: retorna `true` se o primeiro operando é menor que o segundo operando; caso contrário, retorna `false`.
- >: retorna `true` se o primeiro operando é maior que o segundo operando; caso contrário, retorna `false`.
- <=: retorna `true` se o primeiro operando é menor ou igual que o segundo operando; caso contrário, retorna `false`.
- >=: retorna `true` se o primeiro operando é maior ou igual que o segundo operando; caso contrário, retorna `false`.

Vejamos alguns exemplos, reproduza-os em seu `console`:

```
1 11 > 3;         //retorna: true
2 8 < 10;         //retorna: true
3 12 >= 71;       //retorna: false
4 -13 <= -9;      //retorna: true
5 9 > 9;          //retorna: false
6 55 <= 8;        //retorna: false
```

Bem simples, né!? Bora lá que tem mais...

4.1.5 Operadores de lógicos

Os operadores lógicos `&&` (E lógico), `||` (OU lógico) e `!` (NÃO lógico) efetuam álgebra booleana e são frequentemente usados em conjunto com os operadores relacionais para combinar duas expressões relacionais em outra mais complexa. Antes de vermos esses operadores em ação apresento na Figura 4.2 as tabelas-verdade de cada um deles.

Veremos um exemplo prático disso. Suponha que uma proposição seja $5 > 10$ (`A`) e outra proposição seja $9 == 9$ (`B`). Sabemos que `A` é falso e `B` é verdadeiro, certo? Logo, `A` E `B` será falso e `A` OU `B` será verdadeiro. Esses operadores serão usados, principalmente, se quisermos realizar expressões lógicas como saber se um número esta entre dois valores ($120 \geq x \geq 10$).

Vamos ver mais alguns exemplos com *javascript*:

```
1 var x = 50;
```

```
2  x >= 10;                    //retorna: true
3  x <= 120;                   //retorna: true
4  (x >= 10) && (x <= 120);    //retorna: true
5  (x >= 80) && (x <= 120);    //retorna: false
6  (x >= 80) || (x <= 120);    //retorna: true
7  !(x <= 120);                //retorna: false
8  !(x > 120);                 //retorna: true
9  !(x < 50);                  //retorna: true
```

Observe que para o `&&` retornar `true` as duas proposições (operadores) devem ser `true`. Por isso, na linha 4 temos o retorno `true`, é o caso da verificação $120 \geq 50 \geq 10$ (50 é maior ou igual a 10 E 50 é menor ou igual a 120); e na linha 5 temos o retorno `false`, $120 \geq 50 \geq 80$ (50 é maior ou igual que 80 E 50 é menor ou igual a 120). Na linha 6 temos um retorno `true`, pois precisamos que pelo menos uma das proposições seja verdadeira. E, na linha 7 temos uma negação de algo `true` que retorna `false`.

A	B	A **OU** B
V	V	V
V	F	V
F	V	V
F	F	F

A — Tabela verdade do **OU** lógico, retorna verdadeiro sempre que pelo menos uma das proposições for verdadeira.

A	B	A **E** B
V	V	V
V	F	F
F	V	F
F	F	F

B — Tabela verdade do **E** lógico, retorna verdadeiro somente se as duas proposições for verdadeira.

A	**NÃO** B
V	F
F	V

C — O **NÃO** lógico, retorna sempre o oposto do valor lógico da proposição.

Figura 4.2: Tabela-verdade envolvendo os operadores de E lógico, OU lógico e NÃO lógico. Cada tabela exibe todas as combinações possíveis das proposições `A` e `B`.

4.1.6 Operadores de atribuição

Além do operador de atribuição normal (=), o *javascript* aceita vários outros operadores de atribuição que fornecem atalhos por combinar atribuição com alguma outra operação. Por exemplo: o operador `-=` efetua subtração e atribuição. A expressão a seguir:

```
total -= impostos
```

é equivalente a esta:

```
total = total - impostos.
```

Na Tabela 4.1 mostro uma lista das operações de atribuição possíveis, sua equivalência e exemplos. O uso desses operadores servem para simplificarmos mais o código.

! Os operadores de atribuição só podem ser utilizados com variáveis.

Não se preocupe se você não entendeu tudo. Este material servirá de referência para quando precisar. Ao longo dos exemplos não utilizaremos nem 50% desses operadores. Faremos mais alguns exemplos com expressões compostas, após vermos os comandos de entrada e saída de dados.

Operador	Exemplo	Equivalente	Estudo de Caso

+=	a+=b	a = a + b	`var a = 5;`
			`var b = 10;`
			`a += b;  //a é 15 e b é 10`

-=	a-=b	a = a - b	`var a = 15;`
			`var b = 10;`
			`a -= b;  //a é 5 e b é 10`

*=	a*=b	a = a * b	`var a = 3;`
			`var b = 5;`
			`a *= b;  //a é 15 e b é 5`

**=	a**=b	a = a ** b	`var a = 5;`
			`var b = 2;`
			`a **= b;  //a é 25 e b é 2`

/=	a/=b	a = a / b	`var a = 5;`
			`var b = 10;`
			`a /= b;  //a é 0.5 e b é 10`

%=	a%=b	a = a % b	`var a = 10;`
			`var b = 7;`
			`a %= b;  //a é 3 e b é 7`

Tabela 4.1: Principais operadores de atribuição.

4.2 Entrada e Saída de Dados

Vimos na Seção 2.2 que é importante termos comandos para entrada e saída de dados (informações), pois fazem parte do objetivo de todo programa: resolver problemas. Todas as representações de algoritmo utilizam esses comandos para obtermos a informação, na entrada (*input*), e após o processamento do programa escrevemos na saída (*output*) o resultado dos cálculos realizados, ou mesmo um texto para notificar o usuário da nossa aplicação do que está ocorrendo internamente no programa.

Como comecei a falar de programa vamos começar a trabalhar com outra área do Replit, a área onde colocaremos o código-fonte (Figura 3.2E). Os comandos que falarmos nesta seção serão colocados nessa área.

4.2.1 Entrada de Dados no Javascript

A entrada dos dados no *javascript* será realizada por meio do comando `prompt()`. Opcionalmente, você pode adicionar uma `string` entre os parênteses do comando para informar ao usuário o que você quer. Por exemplo: `prompt("Informe seu nome");` será exibido no `console` o texto `Informe seu nome>` e o `console` aguarda que você escreva algo. Ao usar o `prompt()` em nosso programa teremos que atribuir sua execução a uma

variável, somente, assim, realmente iremos capturar a informação dada pelo usuário e, posteriormente, realizar operações com este dado.

Um detalhe importante relacionado ao `prompt()` é que independentemente do valor informado ele sempre irá convertê-lo para uma `string`, logo se quisermos capturar um `number` deveremos converter a informação capturada pelo `prompt()` de forma apropriada. Vejamos algumas conversões de tipo de dados, também chamadas de *casting*, que serão bastante úteis:

- `String(x)` ou `(x).toString()`: converte o dado `x` para uma `string`. Se `x` for um `boolean` a `string` será seu valor em forma de texto, `"true"` ou `"false"`.
- `Number(x)`: converte um dado `x` para o `number`; caso `x` seja uma `string` que possua algo além de números a conversão retorna o valor `NaN`, representa um valor inválido. Caso `x` seja um `boolean`, o valor retornado será `1` caso seja `true` e `0` caso seja `false`.
- `parseInt(x)`: este comando deve ser usado no caso em que você queira garantir que `x` seja transformado em um número inteiro.
- `parseFloat(x)`: este comando deve ser usado no caso em que você queira garantir que `x` seja transformado em um número real.

Irei limitar nossos estudos a esses comandos de conversão, pois serão suficientes para fazermos nossas práticas e exercícios, porém, existem outros tipos de conversão[1]. Você deve estar se perguntando, cadê os exemplos? Vamos ver os exemplos após eu mostrar os comandos de saída, aí as coisas irão fluir melhor.

4.2.2 Saída de Dados no Javascript

A saída de dados, como falamos na Seção 2.2, é a impressão de uma informação. Isso no *javascript* será realizado por meio do comando `console.log(x)`, onde `x` é a informação que queremos que apareça em nosso `console`. Este comando é bastante versátil, pois em `x` você pode usar qualquer tipo de dado, e caso queira imprimir informações diferentes você pode concatená-las ou utilizar a vírgula. Vejamos o Código 4.1:

Código 4.1 Saída de Dados Simples[2]

```
1 var a = 10;
2 var b = 7;
3 var soma_a_b = a+b;
4 console.log(a + "+" + b + "=" + soma_a_b)  //será impresso: 10+7=17
```

■

Para executar o Código 4.1, e os demais códigos que iremos ver a partir de agora, iremos colocar este comando na área de código-fonte e clicar no botão de executar (Figura 3.2B), o resultado aparecerá no `console`. Na Figura 4.3 mostro o passo a passo.

No Código 4.2 irei mostrar outras formas de utilizar o `console.log()` que produzirão o mesmo resultado.

Código 4.2 Saída de Dados Alternativas[3]

```
1 var a = 10;
2 var b = 7;
3 var soma_a_b = a+b;
4 console.log(a,"+",b,"=",soma_a_b)          //será impresso: 10 + 7 = 17
5 console.log(`${a} + ${b} = ${soma_a_b}`)   //será impresso: 10 + 7 = 17
```

[1] <https://www.w3schools.com/js/js_type_conversion.asp>
[2] <https://replit.com/@DaniloBorges/SaidaDeDados-Ex1>
[3] <https://replit.com/@DaniloBorges/SaidaDeDados-Ex2>

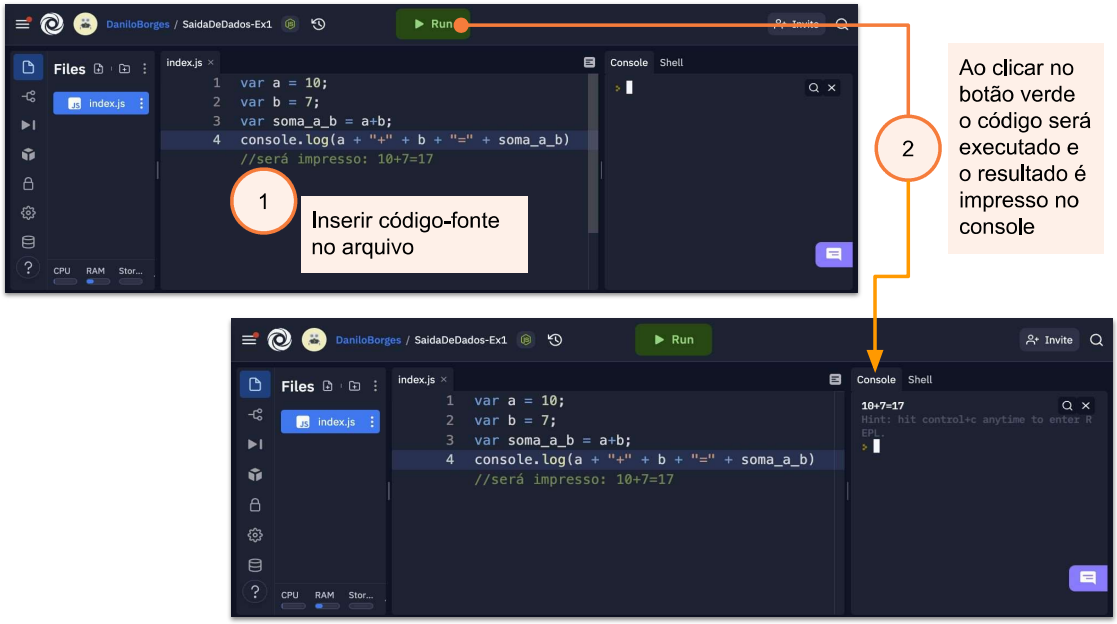


Figura 4.3: Executando o código. 1) O código-fonte fica nesta área. 2) O processo de executar o código é realizado botão verde (*Run*), o seu programa irá processar da linha 1 até a linha 4, e se houverem comandos de entrada e saída de dados esses valores aparecerão no `console`.

Na linha 4 temos cada termo que queremos exibir separado por vírgulas; na saída o comando retorna uma `string`, onde cada termo é convertido em uma `string` e a vírgula é substituída por um espaço, na concatenação. Na linha 5 temos um exemplo do poder da definição de uma `string` utilizando crase. Podemos escrever uma `string` normalmente e no local onde queremos que o valor da variável apareça a colocamos entre chaves com um cifrão junto a abertura da chave, (`${<variável>}`). ■

4.3 Combinando Comandos de Entrada e Saída de Dados no Javascript

Para ilustrar o uso dos comandos de entrada e saída de dados, antes de cada código apresentado falarei o que será realizado.

■ **Código 4.3** Entrada e Saída de Dados para Soma[4]. Neste código são obtidas duas informações pelo console que serão convertidas para `number` e somadas. Posteriormente, é exibido cada um operando e o resultado da soma.

```
1 console.log("== Programa que soma dois números == ");
2 var a = Number(prompt("Informe o primeiro número"));
3 var b = parseFloat(prompt("Informe o segundo número"));
4 var soma_a_b = a+b;
5 console.log("Resultado:")
6 console.log(`${a} + ${b} = ${soma_a_b}`)
```

Na Figura 4.4 mostro o passo a passo da execução deste código. No exemplo, o usuário coloca dois números `80` e `78` e o programa imprime a soma dos mesmos: `158`. ■

Os próximos dois códigos não serão ilustrados, pratique e veja os resultados no `console`.

[4] <https://replit.com/@DaniloBorges/EntradaSaidaDeDados-Ex1>

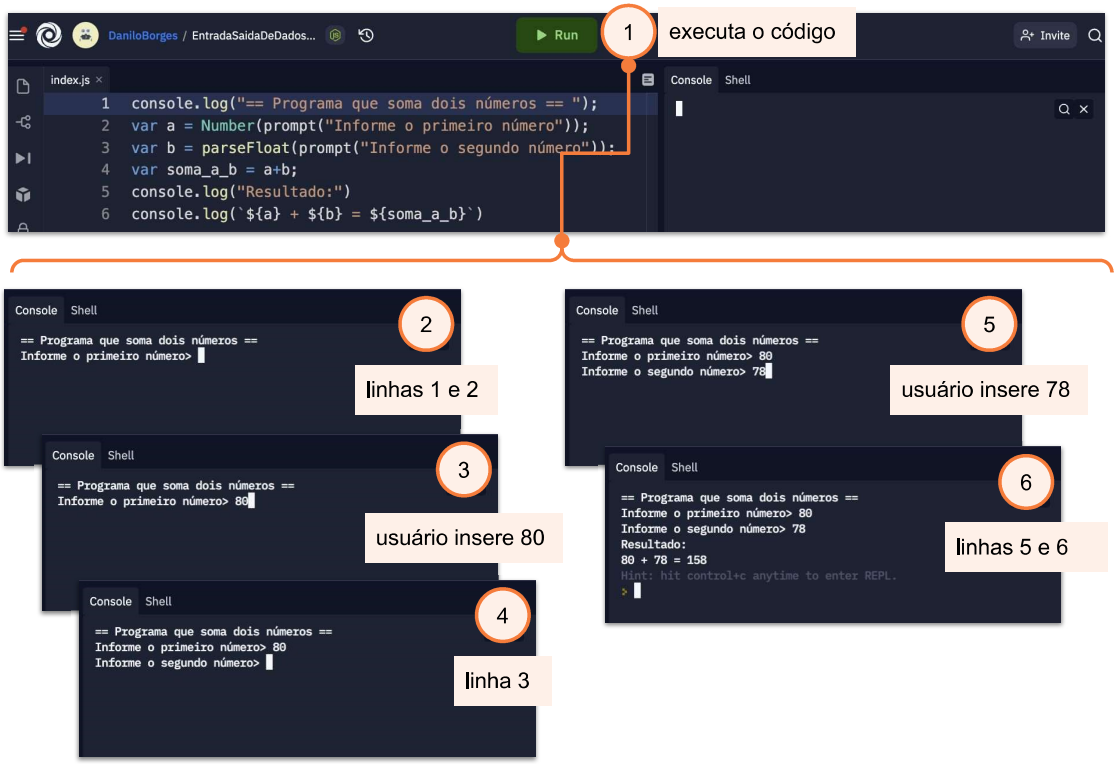


Figura 4.4: Execução do Código 4.5. A sequência indica como cada resultado irá aparecer no console, primeiramente deve-se executar o código. As linhas indicadas no texto se referem as linhas que são processadas no código-fonte para produzir a saída no console.

■ **Código 4.4** Entrada e Saída de Dados para Apresentação[5]. Neste código é obtido o nome do usuário e sua idade, e é posteriormente impresso essa informação no `console`.

```
1 console.log("== Programa apresentação == ");
2 var nome = prompt("Informe seu nome");
3 var idade = parseInt(prompt("Informe a sua idade"));
4 console.log(`Olá, ${nome}! Você tem ${idade} anos.`);
```

■

■ **Código 4.5** Entrada e Saída de Dados para Ano Bissexto[6]. Neste código é obtido um ano (2020, 2022, etc.), e, posteriormente, é impresso no `console` qual seria a possibilidade do ano informado ser bissexto.

```
1 console.log("== Programa verifica bissexto == ");
2 var ano = parseInt(prompt("Informe ao ano"));
3 var verifica = ano%4;
4 console.log(`Para o ano ser bissexto ${ano} módulo 4 deve ser igual a
      zero.`);
5 console.log("O resultado foi: ",verifica);
```

■

Legal, começamos a desenvolver uma parte essencial para os nossos programas: uso de operadores e de comandos de entrada e saída. A próxima etapa será conhecer e utilizar as

[5] <https://replit.com/@DaniloBorges/EntradaSaidaDeDados-Ex2>
[6] <https://replit.com/@DaniloBorges/EntradaSaidaDeDados-Ex3>

estruturas de alteração de fluxo na Parte III.

4.4 Exercícios

Exercício 4.1 Marque quais comandos de conversão que podemos utilizar para converter de `string` para `number`:

a) `Number()`
b) `console.log()`
c) `parseInt()`
d) `parseFloat()`

■

Exercício 4.2 Quais dos operadores abaixo podem ser tanto unários quanto binários:

a) `+`
b) `-`
c) `**`
d) `%`
e) `/`

■

Exercício 4.3 Quais dos operadores abaixo retornam um valor lógico, `boolean`:

a) `-`
b) `&&`
c) `==`
d) `!`
e) `>=`

■

Exercício 4.4 Utilize o `console` do javascript para realizar as seguintes operações:

a) $3-5^2$. *Resposta esperada:* `-22`
b) $\frac{25}{7}-\sqrt{25}$. Dica: na raiz quadrada você deve elevar o número para `0.5`. *Resposta esperada:* `-1.4285714285714284`
c) $\frac{3}{7}\times 5-3\times(-2+5^{-1})$. *Resposta esperada:* `7.542857142857143`
d) $8\%3==5$. *Resposta esperada:* `false`
e) $x>2$ `E` $x\leq 5$, onde $x=7$. *Resposta esperada:* `false`
f) $(x>2)$ `OU` $(x>5$ `E` $x\leq 6)$, onde $x=3$. *Resposta esperada:* `true`

■

Exercício 4.5 Faça um código que realiza a multiplicação de dois números e imprima o resultado. O usuário irá informar os dois números. Dica: utilize como base o Código 4.5.. ■

Exercício 4.6 Faça um código que recebe três números (`a`, `b` e `x`) e imprima o resultado da expressão `a*x+b`. O usuário deverá informar os três números. ■

III

Alterando o Fluxo do Programa

5. Estruturas de Decisão

Neste capítulo você aprenderá a:

- Transformar um algoritmo que possua decisão em um programa em *javascript*;
- Utilizar comandos de se-senão no *javascript*;
- Utilizar comandos de seleção múltipla no *javascript*.

5.1 Mudança de Fluxo

Vamos começar a fazer mudança no fluxo dos nossos problemas. A mudança que me refiro é que usualmente todo programa irá executar linha por linha do código-fonte, porém existem comandos que fazem com que o programa "pule linhas", ou seja, eles mudam o fluxo linear do programa. Fizemos um algoritmo que possui essa propriedade no Exemplo: 2.6, aplicado ao Problema 2.2.3. Onde conforme o resultado da decisão o fluxo seguia uma direção diferente.

Nas representações de fluxograma e pseudocódigo fica evidente que existe uma mudança de fluxo que ocorre com base na condição de um **se** (Exemplo 2.6 e Exemplo: 2.9). Veremos como transformá-las em um programa em *javascript*. Sem mais delongas vamos conhecer as seguintes estruturas de decisão: **se**, **se-senão**, **se-senão se-senão** e **caso-selecione**. Cada uma das estruturas será representada com um fluxograma para melhor assimilação dos conceitos e exemplos. Vamos lá!

5.2 Estrutura Se

Toda estrutura de decisão que possui um **se** terá que verificar uma condição. Essa condição, proposição, deverá ser verdadeira para que o escopo (conjunto de instruções) do **se** seja executado. Observe o fluxograma na Figura 5.1, o utilizaremos como base para os exemplos. Mas, primeiro darei um exemplo de uso mais prático.

Você está com sede e está indo para a casa de um amigo. Chegando lá você espera que lhe ofereçam água, se te oferecem água você bebe e mata sua sede, caso não aconteça você continua com sede. Agora vamos associar isso com a nossa estrutura **se**. Caso te

ofereçam água a condição foi satisfeita você entrou na estrutura do **se** e bebeu, porém se não acontecer você continua na mesma (com sede), o **se** não foi executado. Entendeu!?

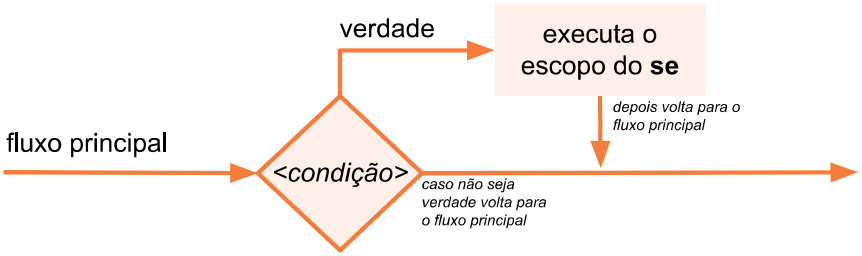


Figura 5.1: Fluxograma da estrutura do **se**. Caso a condição do **se** seja verdade o escopo (conjunto de instruções) internas será executado, caso contrário o programa segue o fluxo normal.

Fiz o exemplo da sede em um fluxograma (Figura 5.2). Com base nele faremos o código em *javascript*. Vamos primeiro vamos ver a estrutura **se** no *javascript*:

```
1  if (<condição>) {
2      <instruções>
3  }
```

A palavra-chave que vai identificar essa estrutura é o `if`, após você adicioná-lo entre parênteses, informe a condição. Lembre-se que a condição tem que ser algo que vai retornar `true` ou `false`. Entre chaves você irá colocar todas as instruções que você quiser executar, ou seja, as chaves definem o tal do escopo. Pronto, está aí o nosso **se** no *javascript*. Vamos ao código do exemplo da sede (Código 5.1).

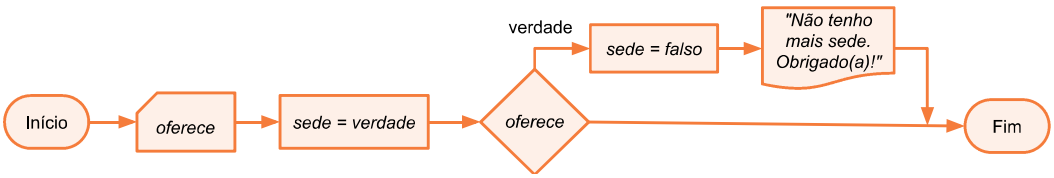


Figura 5.2: Fluxograma da estrutura do **se** no estudo de caso da sede.

■ **Código 5.1** Sede[1] (Fluxograma da Figura 5.2). Para ficar mais dinâmico você irá escrever, no `console`, `Sim`, se seu amigo oferecer água e não escreve nada (dê um enter) caso seu amigo não tiver oferecido.

```
1  var oferece = Boolean(prompt("Seu amigo ofereceu água?"));
2  var sede = true;
3  if (oferece){
4    sede = false;
5    console.log("Não tenho mais sede. Obrigado(a)!");
6  }
```

O `Boolean` é um conversor, como o `Number`. Sua ação é transformar um texto em `boolean`. Caso um texto for vazio ele retorna `false`, caso contrário ele retorna `true`.

■

[1] <https://replit.com/@DaniloBorges/EstruturaSe-Ex1>

No Código 5.1, quando executá-lo você não verá nada, caso não tenha escrito algo no `console` (dado um *enter*). Por isso, não se assuste. Agora se você digitar algum texto irá aparecer a mensagem de agradecimento.

Convenhamos que tem gente que não tem bom trato com as visitas né... então, vamos ter que pedir a água caso não nos ofereçam.

5.3 Estrutura Se-Senão

Essa estrutura também precisa de um **se** com uma condição, porém, caso ela não seja satisfeita podemos fazer outra ação. Sendo assim, ao existir um **senão** garantidamente um dos escopos será executado, o escopo do **se**, ou o escopo do **senão**. Vejamos o fluxograma dessa estrutura na Figura 5.3.

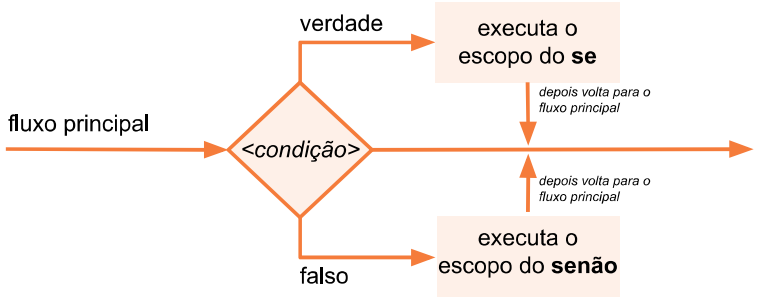


Figura 5.3: Fluxograma da estrutura do **se-senão**. Caso a condição do **se** seja verdade o escopo do **se** será executado, caso contrário o escopo do **senão** será executado. Não importa qual será executado, o fato é que um deles será e depois o fluxo principal é retomado.

Agora podemos fazer o seguinte, se não (**senão**) nos oferecerem água podemos pedir um copo com água. Vamos fazer um código similar ao que o fluxograma da Figura 5.4.

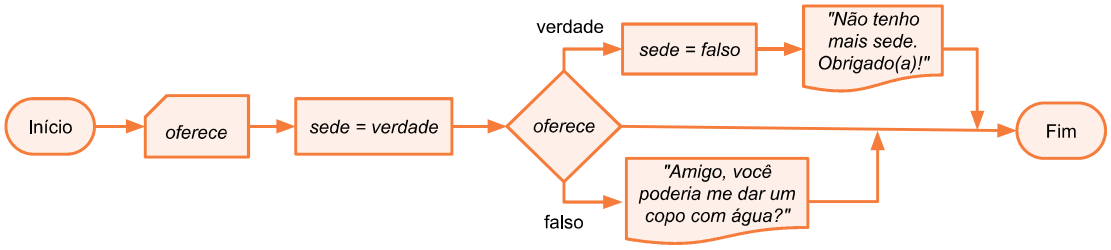


Figura 5.4: Fluxograma da estrutura do **se-senão** no estudo de caso da sede.

A estrutura do **se-senão** começa com a estrutura do **se**, no final do escopo do `if` adicionamos a palavra-chave `else`, que indica o nosso **senão**:

```
1 if (<condição>) {
2     <instruções>
3 }else{
4     <instruções>
5 }
```

Observe que o `else`, também, possui a definição do escopo, as chaves, todas as instruções que tiverem dentro serão executadas caso a `condição` não for satisfeita. Vamos ver o código do exemplo da sede usando essa nova estrutura:

■ **Código 5.2** Sede com Senão[2] (Fluxograma da Figura 5.4). Esse código é similar ao Código 5.1, a diferença está na presença do `else`.

```
1 var oferece = Boolean(prompt("Seu amigo ofereceu água?"));
2 var sede = true;
3 if (oferece){
4   sede = false;
5   console.log("Não tenho mais sede. Obrigado(a)!");
6 }else{
7   console.log("Amigo, você poderia me dar um copo com água?")
8 }
```

■

Agora com este `else`, caso seu amigo não lhe ofereça água você pode pedir. O `else` NÃO existe sem a existência do `if`, note que o colocamos no final do escopo do `if`, depois do encerramento da chave (`...}else`, linha 6). E, só podemos ter um único `else` após o `if`.

Agora podemos, também, fazer o código do Exemplo 2.6, que usa uma estrutura de **se-senão**:

■ **Código 5.3** Raiz da Equação de Primeiro Grau[3].

```
1 var a = Number(prompt("Informe o valor de a"));
2 var b = Number(prompt("Informe o valor de b"));
3 if(a==0){
4   console.log("Não existe raiz");
5 }else{
6   var x = -b/a;
7   console.log(`A raiz da equação ${a}x+(${b}) é ${x}`);
8 }
```

■

O `else` é opcional, você irá utilizá-lo sempre que você ver a necessidade de que alguma coisa deve acontecer se a condição do `if` não for satisfeita (verdadeira).

5.4 Estrutura Se-Senão Se-Senão

Tranquilo até aqui? Espero que sim. Vamos ver agora mais uma possibilidade de mudar o fluxo do nosso programa. Ocorrem casos em que queremos verificar várias situações antes de decidir o que fazer. Vejamos um exemplo disso no contexto universitário.

Usualmente, existem critérios para aprovação e reprovação do(a) aluno(a) mediante a sua nota, e mediante essa nota ainda pode ter uma chance de aprovação caso obtenha uma nota boa na avaliação final (AF). Vou explicar como pode ocorrer isso, passo a passo:

1. Calcula-se a média aritmética de três notas do(a) aluno(a);
2. Se a média for maior ou igual a 7, o(a) aluno(a) estará aprovado(a) na disciplina;
3. Senão se a média for maior ou igual a 4 e menor que 7, o(a) aluno(a), terá que fazer a AF da disciplina;
4. Senão o(a) aluno(a), estará reprovado(a) na disciplina.

Observe que apareceu ali o texto "Senão se", é neste conceito que trabalharemos agora. Vejamos o fluxograma desta estrutura na Figura 5.5. Observa-se que os **se** abaixo só serão executados se o **se** anterior não for verdade, caso qualquer um dos **se** for verdadeiro os

[2] <https://replit.com/@DaniloBorges/EstruturaSeSenao-Ex1>
[3] <https://replit.com/@DaniloBorges/EstruturaSeSenao-Ex2>

posteriores não serão executados. Em resumo, se você quiser verificar várias condições onde só uma delas você quer que seja satisfeita recorra a esta estrutura.

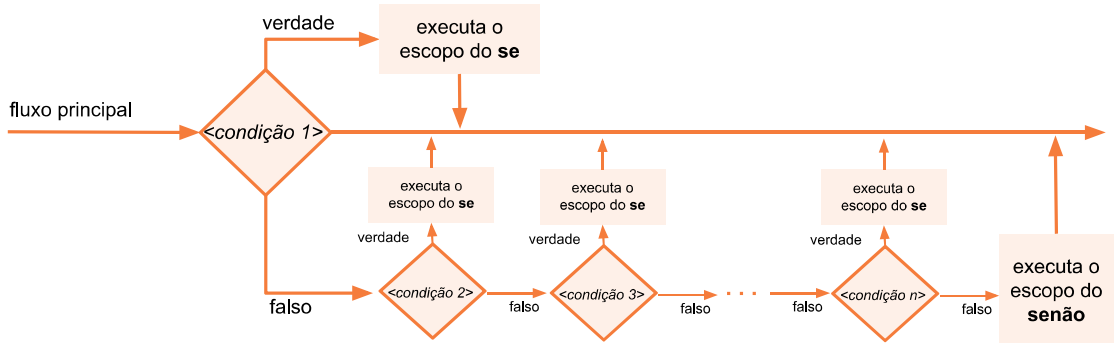


Figura 5.5: Fluxograma da estrutura do **se-se senão-senão**. Caso a condição de algum **se** seja verdade o escopo deste **se** será executado, caso nenhuma condição seja satisfeita o escopo do **senão** será executado.

Vejamos como fica o código em **javascript** da estrutura **se-se senão-senão**:

```
1 if (<condição>) {
2     <instruções>
3 }else if (<condição>){
4     <instruções>
5 }else{
6     <instruções>
7 }
```

Para ilustrar coloquei somente um **senão se**. Você pode colocar quantos `else if` (linha 3) quiser, sempre com o escopo definido (chaves); mas tenha cuidado em relação à posição do `else` sozinho, ele só irá aparecer no final. O `else` como disse é opcional, logo, se for necessário e dependendo do problema, você não precisa colocar o `else`.

Na Figura 5.6, temos o fluxograma do problema da nota. E, no Código 5.4 temos seu funcionamento em *javascript*.

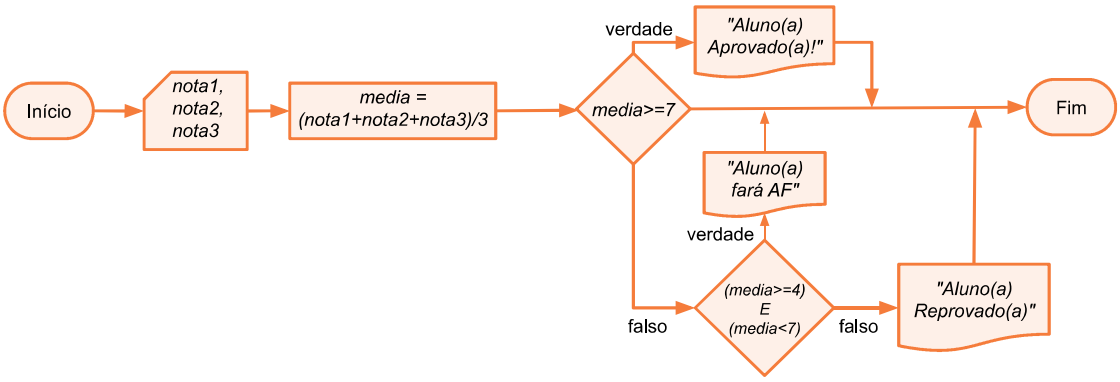


Figura 5.6: Fluxograma da estrutura do **se-senão se-senão** do exemplo da nota.

O código 5.4 pode ser modificado para outra verificação no comando `else if`. Por exemplo: você poderia alterar a condição para `media<4` e colocar o texto de impressão que o aluno está reprovado e no `else` colocaria que está de AF. Ou seja, você pode livremente

mudar as condições. Porém, tenha cuidado para não colocar condições que são satisfeitas em mais de um `if`, o ideal é que cada condição esteja associada somente a um dos escopos.

■ **Código 5.4** Resultado da Média[4].

```
1  var nota1 = Number(prompt("Insira a primeira nota"));
2  var nota2 = Number(prompt("Insira a segunda nota"));
3  var nota3 = Number(prompt("Insira a terceira nota"));
4  media = (nota1+nota2+nota3)/3;
5  if(media>=7){
6    console.log("Aluno(a) Aprovado(a)");
7  }else if((media>=4) && (media<7)){
8    console.log("Aluno(a) fará AF");
9  }else{
10   console.log("Aluno(a) Reprovado(a)");
11 }
```

■

Em resumo, sempre que você quiser fazer um teste (**se**) você usará o `if`, se já existir um `if` você terá que colocar um `else if`. Caso queira recorrer a um escopo se nenhuma condição for satisfeita você coloca o `else`. Cada um dos três comandos devem possuir a abertura e fechamento das chaves. Beleza!?

5.5 Estrutura Caso-Selecione

Esta estrutura difere dos nossos comandos de **se**, mas é bem simples. Observe o fluxograma na Figura 5.7. Nesta estrutura é verificada se acontece uma igualdade da variável em uma lista de opções. Então, é verificado se a variável é igual a um determinado valor, x, se for, o escopo será executado; caso nenhum seja executado, opcionalmente, você poderá colocar um escopo final. Esse escopo final é como se fosse o `else` do `if`.

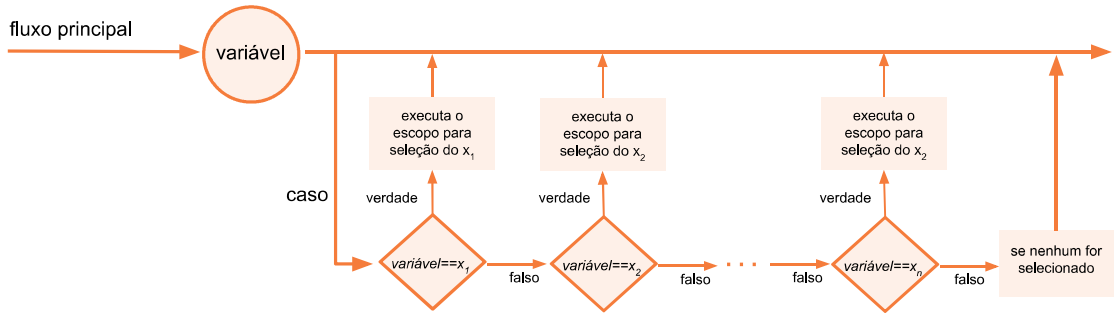


Figura 5.7: Fluxograma da estrutura do **caso-selecione**. Aqui temos uma nova representação de fluxograma o círculo que indica o **caso**.

Para exemplificar vamos trabalhar com o seguinte exemplo: A pessoa vai jogar um jogo chamado *jokenpo*. Ela deverá selecionar uma das seguintes opções: 1, para ser pedra (✊); 2, para ser papel (✋); e 3, para ser tesoura (✌). O programa deve informar essas opções e depois informar qual opção o usuário selecionou. Caso não tenha colocado um dos três valores o programa deve dizer que não foi informado uma opção válida. Baseado nisso temos o fluxograma deste problema na Figura 5.8.

[4] <https://replit.com/@DaniloBorges/EstruturaSeSeSenaoSe-Ex1>

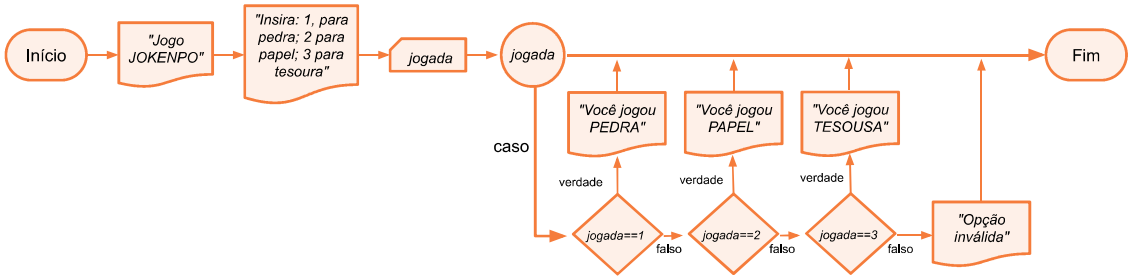


Figura 5.8: Fluxograma da estrutura do **caso-selecione** do exemplo do *jokenpo*.

No fluxograma (Figura 5.8) observe que vamos testar uma variável entre uma lista de opções, selecionamos a opção na qual a igualdade é satisfeita, caso isso não aconteça caímos na última instrução, que é imprimir que a opção é inválida. Vejamos agora a estrutura deste comando no *javascript*, chamada de `switch`:

```
1  switch(<variável>){
2    case <valor_1>:{
3      <instruções>
4      break
5    }
6    case <valor_2>:{
7      <instruções>
8      break
9    }
10   default :{
11     <instruções>
12   }
13 }
```

Coloquei aqui somente duas seleções (linhas 2 e 6), mas você pode colocar quantas quiser. Cada escopo da seleção irá definir as instruções que você quer executar e no final você deve colocar o comando `break`. Esse comando é responsável por parar o fluxo o `switch`. Observe que existe uma opção que não tem um `case`, na linha 10 temos o `default`. O `default` será executado sempre que não houver nenhuma seleção válida e sempre ficará por último. Legal, né!? Já dá para imaginar muitas aplicações desta estrutura. Vamos ao Código 5.5.

■ **Código 5.5** Jogada no Jokenpo[5].

```
1  console.log("Jogo JOKENPO");
2  var jogada = Number(prompt("Insira: 1, para pedra; 2, para papel; 3,
       para tesoura"));
3  switch(jogada){
4    case 1:{
5      console.log("Você jogou PEDRA");
6      break;
7    }
8    case 2:{
9      console.log("Você jogou PAPEL");
10     break;
11   }
12   case 3:{
13     console.log("Você jogou TESOURA");
14     break;
```

[5] <https://replit.com/@DaniloBorges/EstruturaCasoSelecione-Ex1>

```
15    }
16    default:{
17      console.log("Opção inválida");
18    }
19  }
```

■

O `switch` é um comando que permite com que você possa minimizar o código, visto que com a estrutura de `if`, `else if` e `else` pode-se tranquilamente fazer o mesmo programa, porém levariam mais linhas de código.

5.6 Resumo

São muitas estruturas e com a prática você irá aprendendo quais utilizar em cada situação. Recomendo que você faça os exercícios e pratique tudo o que vimos até aqui. Coloquei na Tabela 5.1 os comandos em *javascript* com a descrição de uso.

Comando	Descrição
`if(<condição>)`	Corresponde ao **se**, você deve colocar em `<condição>` uma expressão que retorna um valor booleano, caso seja verdade o escopo do **se** será executado.
`else if(<condição>)`	Essa estrutura avalia se uma outra `<condição>` é satisfeita. Este comando só pode aparecer depois de um `if`.
`else`	Essa estrutura informa se caso nenhum `if`, ou `else if`, for satisfeito (suas condições) deverá ser executado seu escopo.
`switch(<variável>)`	Corresponde a inicialização da estrutura do **caso-selecione**. Esse comando irá avaliar uma determinada `<variável>` dentro de seu escopo constituido por `case`s.
`case <valor>`	Avalia, dentro do escopo do `switch`, se a `<variável>` é igual a `<valor>`. Caso seja verdade seu escopo é executado.
`break`	Comando que para a execução do `switch`. Deve ser colocado no final do escopo de cada `case` do `switch`.
`default`	Esse comando é colocado no final do escopo do `switch` como alternativa caso nenhum `case` seja satisfeito.

Tabela 5.1: Resumo dos comandos das estruturas de seleção em *javascript*.

5.7 Exercícios

Exercício 5.1 Faça um programa que leia os valores A, B, C e imprima na tela se a soma de A + B é menor que C. Dica: utilize a estrutura **se-senão**. ■

Exercício 5.2 Faça um algoritmo que leia dois valores inteiros A e B se os valores forem iguais deverá se somar os dois, caso contrário multiplique A por B. Ao final de qualquer um dos cálculos deve-se atribuir o resultado para uma variável C e mostrar seu conteúdo

no `console`. ■

Exercício 5.3 Faça um algoritmo que leia o nome, o sexo e o estado civil de uma pessoa. Caso estado civil seja `"CASADO"` ou `"CASADA"`, solicitar o tempo de casado(a) (anos). Dica: utilize a estrutura **se**. ■

Exercício 5.4 Faça um algoritmo para receber um número qualquer e informar na tela se é par ou ímpar. Dica: utilize o operador de módulo e a estrutura **se-senão**. ■

Exercício 5.5 Faça um algoritmo que leia uma variável e some 5, caso seja par ou some 8, caso seja ímpar, imprimir o resultado desta operação. Dica: utilize o operador de módulo e a estrutura **se-senão**. ■

Exercício 5.6 Escreva um algoritmo que leia três valores inteiros e diferentes e mostre-os em ordem decrescente. Dica: utilize a estrutura **se-senão se-senão**. ■

Exercício 5.7 Crie um programa que leia a idade de uma pessoa e informe a sua classe eleitoral:

1. não eleitor (abaixo de 16 anos);
2. eleitor obrigatório (entre a faixa de 18 e menor de 65 anos);
3. eleitor facultativo (de 16 até 18 anos e maior de 65 anos, inclusive).

Dica: utilize a estrutura **se-senão se-senão**. ■

Exercício 5.8 O IMC – Indice de Massa Corporal é um critério da Organização Mundial de Saúde para dar uma indicação sobre a condição de peso de uma pessoa adulta. A fórmula é IMC = $peso/(altura)^2$

Elabore um algoritmo que leia o peso e a altura de um adulto e mostre sua condição de acordo com a tabela abaixo.

IMC em adultos	Condição
Abaixo de 18.5	Abaixo do peso
Entre 18.5 e 25	Peso normal
Entre 25 e 30	Acima do peso
Acima de 30	Obeso

■

Exercício 5.9 Elabore um algoritmo que calcule o que deve ser pago por um produto, considerando o preço normal de etiqueta e a escolha da condição de pagamento. Utilize os códigos da tabela a seguir para ler qual a condição de pagamento escolhida e efetuar o cálculo adequado.

Código	Condição de Pagamento
1	À vista em dinheiro ou cheque, recebe 10% de desconto
2	À vista no cartão de crédito, recebe 15% de desconto
3	Em duas vezes, preço normal de etiqueta sem juros
4	Em duas vezes, preço normal de etiqueta mais juros de 10%

■

Exercício 5.10 Criar um programa que leia um número inteiro entre 1 e 7 e escreva o dia da semana correspondente. Caso o usuário digite um número fora desse intervalo, deverá aparecer uma mensagem informando que não existe dia da semana com esse número. ■

Exercício 5.11 Criar um programa que leia um número inteiro entre 1 e 12 e escrever o mês correspondente. Caso o usuário digite um número fora desse intervalo, deverá aparecer uma mensagem informando que não existe mês com este número. ■

6. Estruturas de Repetição

Neste capítulo você aprenderá a:

- Entender a funcionalidade de uma estrutura de repetição;
- Utilizar comandos de repetição em *javascript*;
- Aplicar condições de parada de um fluxo de repetição.

6.1 Repetição de Fluxo

Em muitas situações é interessante que consigamos repetir um fluxo, reaproveitando, assim um código. Seja para repetir tudo, por exemplo: (1) fazer um jogo ter várias partidas, ou (2) fazer a tabuada da multiplicação do 9, onde cada linha o código poderia imprimir o valor da multiplicação do 9 por um número de 1 a 10. A ideia é a seguinte, pensando no exemplo (2), ao invés de colocar um `console.log()` para imprimir cada multiplicação de 9 pelos números de 1 a 10, podemos reaproveitar o que se repete e alterar somente os números que serão multiplicados. Esse fluxo de repetição é usualmente chamado de *laço*.

Vamos estudar três estruturas de repetição que possuem características bem semelhantes: **enquanto**, **faça-enquanto**, **desde que-até**. Da mesma forma que fiz com as estruturas de decisão, irei colocar os fluxogramas para que consiga compreender melhor essas estruturas.

6.2 Estrutura Enquanto

A estrutura **enquanto** tem a característica de verificar uma condição, como no **se**, e caso seja verdade seu escopo será executado. Após finalizar a execução do escopo o fluxo volta para antes da verificação da condição para ser verificado novamente, como ilustra a Figura 6.1. A ideia dessa estrutura é voltar a executar o escopo sempre que a condição for verdadeira, logo, você terá que ter um *critério de parada*, caso contrário seu programa ficará executando a estrutura de repetição indefinidamente. Falarei desse *critério* em um exemplo.

Vamos trabalhar com o exemplo da tabuada de multiplicação de nove, mas vamos deixar mais genérica essa aplicação. O usuário irá informar qual número ele quer que o

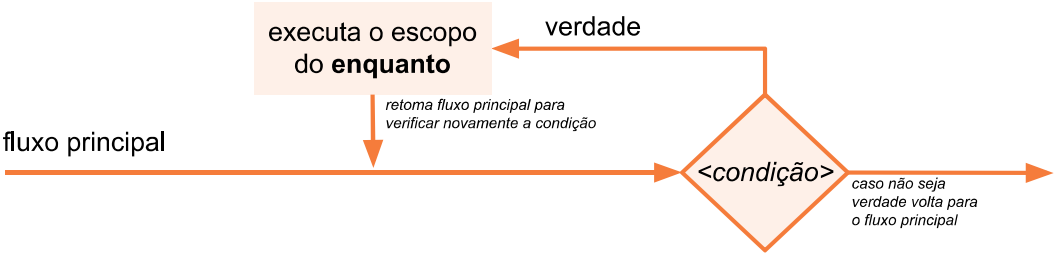


Figura 6.1: Fluxograma da estrutura do **enquanto**.

programa faça a tabuada e, então, mostramos na tela a tabuada de multiplicação deste número pelo número 1 até 10, certo? Vamos ver o fluxograma disso na Figura 6.2.

Observe na Figura 6.2 que a variável `multiplicador` é incrementada dentro do fluxo do **enquanto**. Em um determinado momento o laço será encerrado, ou seja, quando `multiplicador` for igual a 11. Isso é o tal *critério de parada*, um critério no qual a condição do **enquanto** não será satisfeita. Visto que cada critério vai depender, exclusivamente, do seu problema/solução. Certo!?

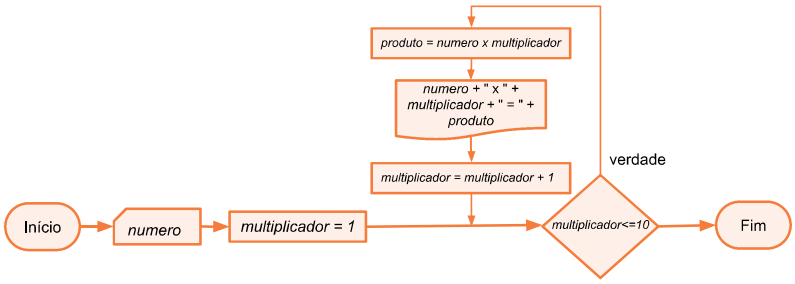


Figura 6.2: Fluxograma da estrutura do **enquanto** do exemplo da tabuada.

Agora vamos ver essa estrutura no *javascript*:

```
1 while (<condição>) {
2     <instruções>
3 }
```

Nosso **enquanto** equivale ao comando `while`, a condição fica logo após o inserirmos no código e as chaves definem o seu escopo. Da mesma forma que nas estruturas condicionais as chaves delimitam o escopo. Bem simples, né? Vejamos agora o código completo da tabuada no *javascript*.

■ **Código 6.1** Tabuada com `while`[1] (Fluxograma da Figura 6.2).

```
1 var numero = Number(prompt("Informe qual tabuada de multiplicação você
      quer (entre 1 e 10)"));
2 var multiplicador = 1;
3 while(multiplicador<=10){
4   produto = numero * multiplicador;
5   console.log(`${numero} x ${multiplicador} = ${produto}`);
6   multiplicador = multiplicador + 1; // pode substituir por
        multiplicador++;
7 }
```

■

[1] <https://replit.com/@DaniloBorges/EstruturaWhile-Ex1>

No Código 6.1 você pode, perfeitamente, substituir a linha 6 por `multiplicador++`, pois produzirá o mesmo efeito. Execute esse código no Replit para ver o funcionamento dessa estrutura.

6.3 Estrutura Faça-Enquanto

Esta estrutura tem um funcionamento similar ao `while` a diferença é que o que desejamos repetir será executado pelo menos uma vez, já no `while` o que queremos repetir só será executado se a condição for verdadeira. Na Figura 6.3, podemos ver que é isso que acontece no **faça-enquanto**. Primeiro ele faz o que queremos, depois ele verifica a condição e se for verdade ele executa o escopo novamente.

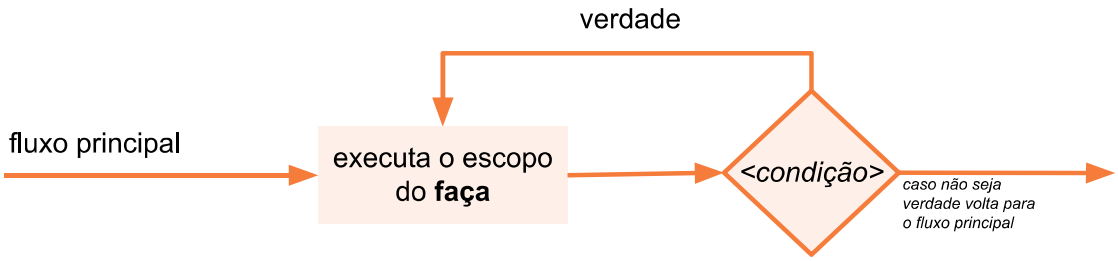


Figura 6.3: Fluxograma da estrutura do **faça-enquanto**.

Essa estrutura no *javascript* também é simples e semelhante ao `while`:

```
1 do {
2     <instruções>
3 } while (<condição>);
```

Observe que o `do` equivale ao nosso **faça**, ele entra direto no escopo e depois avalia se entra novamente no `while` (**enquanto**). Caso seja verdade ele repete o escopo do `do`.

Você pode optar por usar essa estrutura sempre que tiver certeza que um determinado trecho de código deverá ser executado pelo menos uma vez. Na Figura: 6.4 temos o fluxograma da estrutura **faça-enquanto** da tabuada e no Código 6.2 temos o sua implementação.

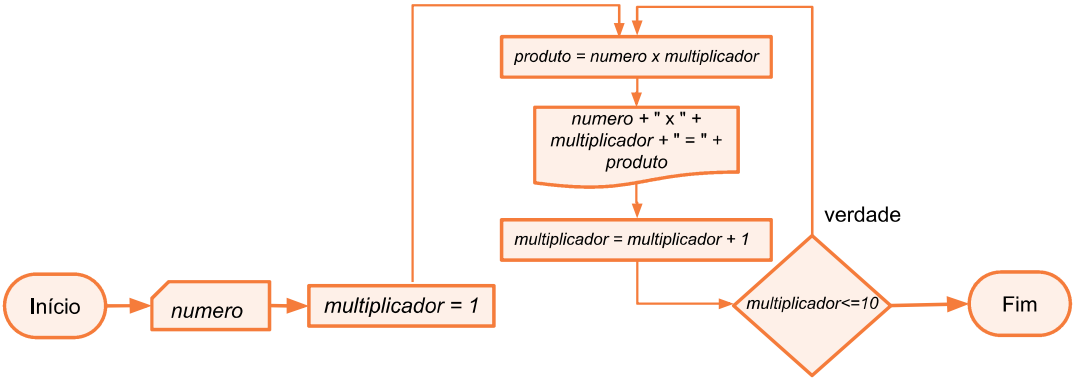


Figura 6.4: Fluxograma da estrutura do **faça-enquanto** do exemplo da tabuada.

■ **Código 6.2** Tabuada com `do-while`[2] (Fluxograma da Figura 6.4).

```
1  var numero = Number(prompt("Informe qual tabuada de multiplicação você
       quer (entre 1 e 10)"));
2  var multiplicador = 1;
3  do {
4    produto = numero * multiplicador;
5    console.log(`${numero} x ${multiplicador} = ${produto}`);
6    multiplicador++;
7  }while(multiplicador<=10);
```

■

No Código 6.2, observe que como não existe nenhuma condição após o `do` o trecho das linhas 4 até 6 é executado. Aí é que a *condição de parada* é verificada, se for verdade o fluxo volta para linha 4, senão a repetição para. Ou seja, pelo menos uma vez o escopo da repetição `do-while` será executado.

6.4 Estrutura Desde que-Até

Essa estrutura é bem interessante, e simplifica muito o trabalho quando temos que incrementar uma variável, como é o caso da tabuada. No fluxograma da Figura 6.5 temos o seu funcionamento.

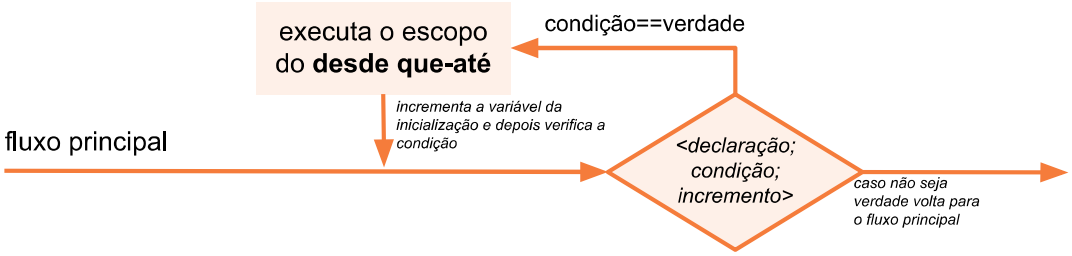


Figura 6.5: Fluxograma da estrutura do **desde que-faça**.

A estrutura **desde que-até** é constituída por três partes: (1) declaração, (2) condição e (3) incremento[3]. Em (1) você irá declarar, e inicializar, a(s) variável(eis) que será(ão) incrementada(s), ou decrementada(s), ao final do escopo; no (2) você irá definir qual condição deve ser satisfeita para entrada no escopo; e em (3) você adiciona o(s) incremento(s) (ou decremento(s)) na(s) variável(eis) declarada(s).

Agora vamos ver o funcionamento dessas três partes na repetição do escopo. Na primeira vez que a estrutura começa sua execução irá ocorrer a inicialização da(s) variável(eis) (1) e o teste da condição (2). Se a condição for satisfeita, então depois de cada laço (escopo), antes de iniciar o escopo novamente, (3) o(s) incremento(s), ou decremento(s), são realizados. O **desde que** para **até** que a condição (2) não seja satisfeita. Logo, já dá para perceber que pode ser que o escopo do **desde que-até** nunca seja executado, caso a condição já falhe após a inicialização.

Essa estrutura no *javascript* se chama `for`:

```
1  for (<declaração(ões)>;<condição>;<incremento(s) ou decremento(s)>) {
2      <instruções>
3  }
```

[2] <https://replit.com/@DaniloBorges/EstruturaDoWhile-Ex1>

[3] Pode ser colocado também um decremento, ou os dois: uma variável com incremento e outra com decremento.

No `for` temos as três partes da estrutura, cada uma delas separadas por ponto e vírgula. Da mesma forma que o `while`, esse comando só será executado se a condição for verdadeira. A declaração (com inicialização) ocorre uma única vez, e o(s) incremento(s) só acontece no final do escopo; é como se existisse o comando de incremento, ou decremento, no final do laço. A entrada no laço só será feita enquanto a condição for verdadeira.

Vamos lá, ficará bem mais claro ao praticarmos. Na Figura: 6.6, temos o fluxograma da tabuada utilizando essa estrutura. Olha como ficou mais simples...

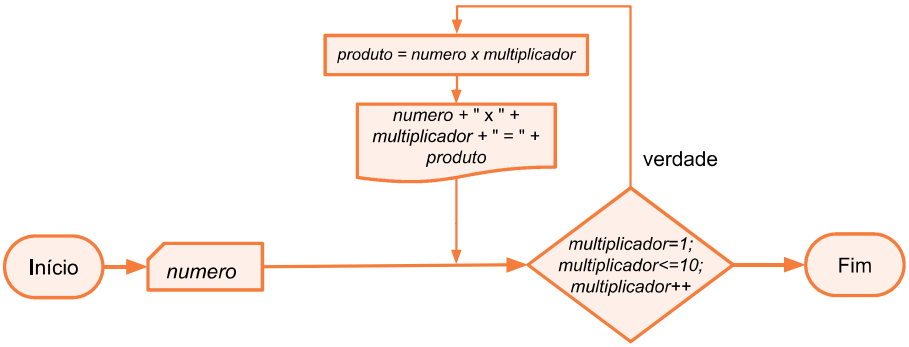


Figura 6.6: Fluxograma da estrutura do **desde que-até** do exemplo da tabuada.

No código 6.3 temos a implementação da tabuada usando essa estrutura. Observe que não precisamos mais declarar o `multiplicador` antes da repetição, colocamos isso na área de declaração (`var multiplicador=1;`); a condição ficou no meio (`multiplicador<=10;`); e no caso dessa aplicação teremos o incremento da variável `multiplicador` que colocávamos no final do escopo (no `while` e `do-while`) agora ficará na estrutura na terceira parte do `for` (`multiplicador++`).

■ **Código 6.3** Tabuada com `for`[4] (Fluxograma da Figura 6.6).

```
1  var numero = Number(prompt("Informe qual tabuada de multiplicação você
       quer (entre 1 e 10)"));
2  for (var multiplicador=1;multiplicador<=10;multiplicador++) {
3    produto = numero * multiplicador;
4    console.log(`${numero} x ${multiplicador} = ${produto}`);
5  }
```

■

Pronto. Mostrei as estruturas básicas de repetição que poderemos usar para solucionar vários problemas. Lembre-se, sempre que for utilizar um desses comandos, de definir o *critério de parada*. Uma outra alternativa para parar o programa, dentro de uma estrutura de repetição, é usar o comando `break`.

6.5 Parar a Repetição

Como havia dito a repetição para quando o *critério de parada* estiver presente, ou seja, quando a condição da repetição for `false`. Porém podemos parar o fluxo da repetição com o comando `break`, similar ao `break` que temos em cada `case` do `switch`. Ao executar esse comando o fluxo para imediatamente.

Para ilustrar esse comportamento farei o seguinte exemplo: o usuário poderá ver quantas tabuadas quiser, porém se ele colocar um número que estiver fora dos números entre 1 e

[4] <https://replit.com/@DaniloBorges/EstruturaFor-Ex1>

10 o programa para. Você já deve ter percebido que vamos trabalhar com uma estrutura condicional para verificar se o número está entre 1 e 10. E com dois laços de repetição: (1) para repetir a impressão da tabuada e (2) para imprimir a tabuada em si. Optei por utilizar em (1) um `do-while` e no (2) um `for`. Vejamos o Código 6.4.

■ **Código 6.4** Tabuada com do Infinito[5]. A palavra infinito se refere a uma repetição na qual a condição é sempre verdadeira, o critério de parada depende totalmente do `break`.

```
1  do{
2    var numero = Number(prompt("Informe qual tabuada de multiplicação voc
         ê quer (entre 1 e 10)"));
3    if(numero<1 || numero>10){
4      console.log(`Você digitou um número inválido: ${numero}`);
5      console.log("Fim do Programa");
6      break;
7    }
8    for (var multiplicador=1;multiplicador<=10;multiplicador++) {
9      produto = numero * multiplicador;
10     console.log(`${numero} x ${multiplicador} = ${produto}`);
11   }
12 } while (true);
```

A estrutura `do` compreende o escopo da linha 2 até a linha 11. E, nessa condição, linha 12, será sempre satisfeita, pois coloquei um `true` dentro dela. Somente se o usuário colocar um número inválido é que essa repetição irá parar, ou seja, se a condição do `if` da linha 3 for satisfeita; será impressa a informação de número inválido, sinalizado o fim do programa, e o `break` irá parar a estrutura de repetição. A repetição feita pelo `for` será responsável por imprimir a tabuada do número informado. ■

Coloquei um exemplo que mostra o uso do `break` bem simples, mas saiba que você pode utilizar sempre que quiser. Na maioria dos usos, dentro de uma repetição, o colocamos dentro de um condicional. Então, se quiser parar a repetição de outra forma, sem ser com a condição, é só utilizá-lo.

6.6 Estudos de Caso

Nesta seção coloquei alguns problemas práticos e os resolvo utilizando estruturas de repetição com *javascript*. Colocarei, também, na definição do problema qual a estrutura mais indicada desenvolver a solução.

Problema 6.6.1 — While ou Do-While ou For. Escreva um programa que encontre o quinto número maior que 1000, cuja divisão por 11 tenha resto 5.

O Problema 6.6.1 pode ser resolvido utilizando qualquer um dos algoritmos de repetição que vimos. O que devemos prestar atenção é no *critério de parada*. Qual seria o *critério de parada* desse problema? Resposta correta: ter encontrado o quinto número maior que 1000, cuja divisão por 11 tenha resto 5. Então, vamos utilizar uma variável para armazenar a quantidade de números que encontrarmos maiores que 1000 cuja divisão por 11 tenha resto 5, e caso tenhamos encontrado cinco números a gente para. Certo?

Nos Códigos 6.5 ao 6.7, chamaremos essa variável de `quantidade_do_criterio` que começa inicialmente com `0`. E, à medida que vamos encontrando os números que queremos vai sendo incrementada.

Você pode colocar então como critério de parada `quantidade_do_criterio<5`, e dentro da repetição caso encontre `quantidade_do_criterio==5` sinalizar que encontrou o tal do

[5] <https://replit.com/@DaniloBorges/EstruturaBreak-Ex1>

quinto número.

Coloquei em cada um dos códigos uma estrutura de repetição diferente, para que você consiga observar as semelhanças e o porquê de cada uma funcionar como esperado.

■ **Código 6.5** Solução do Problema 6.6.1 com `while`[6].

```
1  var numero = 1000;
2  var quantidade_do_criterio = 0;
3  while (quantidade_do_criterio<5){
4    numero++;
5    if(numero%11==5){
6      quantidade_do_criterio++;
7    }
8    if(quantidade_do_criterio==5){
9      console.log("O quinto número maior que 1000, cuja divisão por 11
           tenha resto 5 é",numero);
10   }
11 }
```

■

■ **Código 6.6** Solução do Problema 6.6.1 com `do-while`[7].

```
1  var numero = 1000;
2  var quantidade_do_criterio = 0;
3  do {
4    numero++;
5    if(numero%11==5){
6      quantidade_do_criterio++;
7    }
8    if(quantidade_do_criterio==5){
9      console.log("O quinto número maior que 1000, cuja divisão por 11
           tenha resto 5 é",numero);
10     break;
11   }
12 } while (true);
```

■

■ **Código 6.7** Solução do Problema 6.6.1 com `for`[8].

```
1  var quantidade_do_criterio = 0;
2  for (var numero=1000;quantidade_do_criterio<5;numero++){
3    if(numero%11==5){
4      quantidade_do_criterio++;
5    }
6    if(quantidade_do_criterio==5){
7      console.log("O quinto número maior que 1000, cuja divisão por 11
           tenha resto 5 é",numero);
8    }
9  }
```

■

Problema 6.6.2 — While ou Do-While ou For. Foi feita uma pesquisa entre um grupo de estudantes e coletados os dados de altura e sexo (0=masculino, 1=feminino). Faça um programa que leia 10 dados diferentes e informe:

[6] <https://replit.com/@DaniloBorges/EstudoCaso-Ex1-ComWhile>
[7] <https://replit.com/@DaniloBorges/EstudoCaso-Ex1-ComDoWhile>
[8] <https://replit.com/@DaniloBorges/EstudoCaso-Ex1-ComFor>

1. a maior e a menor altura encontradas;
2. a média de altura dos estudantes de sexo feminino;
3. a média total de altura dos estudantes;
4. o percentual de estudantes do sexo masculino.

Calma, parece muita coisa nesse problema 6.6.2, mas é bem simples de resolver. Primeiro temos que ver como nossa repetição irá agir, você tem alguma sugestão? Vamos agir na quantidade de dados! Caso cheguemos a quantidade de dados (variável `quantidade_dados`) esperada a gente para (`quantidade_dados==10`). Em cada laço da repetição devemos capturar o sexo do estudante (variável `sexo`), e sua altura (variável `altura`). E devemos, também, criar variáveis para armazenar: maior altura (variável `maior_altura`), menor altura (variável `menor_altura`), quantidade de estudantes do sexo masculino (variável `quantidade_masculino`), quantidade de estudantes do sexo feminino (variável `quantidade_feminino`). Com essas variáveis conseguiremos resolver o problema. Embora, seja possível fazer este algoritmo utilizando qualquer estrutura de repetição mostrarei somente a que utiliza o `while`, Código 6.8.

Código 6.8 Solução do Problema 6.6.2 com `while`[9].

```
1  var soma_altura_feminino = 0, soma_altura_masculino = 0;
2  var quantidade_feminino = 0, quantidade_masculino = 0;
3  var altura,sexo;
4  var maior_altura = -Infinity, menor_altura = Infinity;
5  var quantidade_dados = 1;
6  while (quantidade_dados<=10){
7    altura = Number(prompt(`Informe a altura do estudante ${
         quantidade_dados}:`));
8    sexo = Number(prompt(`Informe o sexo do estudante ${quantidade_dados}
         (0 para masculino e 1 para feminino):`));
9    if(sexo==0){ //sexo masculino
10     quantidade_masculino++;
11     soma_altura_masculino += altura;
12   }else if(sexo==1){
13     quantidade_feminino++;
14     soma_altura_feminino += altura;
15   }else{
16     console.log("Sexo inválido, faça novamente.")
17   }
18   if ((sexo==0) || (sexo==1)){
19     if(altura > maior_altura){
20       maior_altura = altura;
21     }
22     if(altura < menor_altura){
23       menor_altura = altura;
24     }
25     quantidade_dados++;
26   }
27 }
28 console.log("Resultado:");
29 console.log("Maior Altura: ",maior_altura);
30 console.log("Menor Altura: ",menor_altura);
31 console.log("Média de altura do sexo feminino",soma_altura_feminino/
       quantidade_feminino);
32 console.log("Média de altura dos estudantes",(soma_altura_feminino+
       soma_altura_masculino)/10);
33 console.log("Sexo masculino corresponde a",((quantidade_masculino)/(
       quantidade_feminino+quantidade_masculino))*100,"%");
```

[9] <https://replit.com/@DaniloBorges/EstudoCaso-Ex2-ComWhile>

■

No código 6.8, você pode observar que inicialmente declarei todas essas variáveis citadas, algumas já com iniciação (linhas 1, 2, 4 e 5). Um caso especial está nas variáveis `maior_altura` e `menor_altura` que foram inicializadas com `-Infinity` e `Infinity`, repectivamente. Mas, porquê coloquei isso aqui? Você acertou quando pensou que não existe esse número. Essa palavra reservada é uma variável, que possui o maior valor possível que o programa pode possuir. Logo, estou inicializando essas variáveis com o menor e maior valor para que na primeira comparação (linhas 19 e 22) elas sejam atualizadas. Uma atenção especial na linha 18, só irei atualizar o contador de dados, e maior altura se foi inserido o código do sexo de forma correta, caso contrário informo que o usuário deve tentar novamente (linha 16). Da linha 28 até a linha 33 temos a exposição dos resultados do que foi requisitado no problema.

Para finalizar nossos estudos de caso, vejamos o Problema: 6.6.3.

Problema 6.6.3 — While ou Do-While ou For. Escreva um programa que lê um valor n inteiro e positivo, e que calcula a seguinte soma:

$$\frac{1}{1}+\frac{2}{3}+\frac{3}{5}+\frac{4}{7}+\cdots+\frac{n-1}{2(n-1)-1}+\frac{n}{(2n-1)}$$

Nesse último problema a estrutura mais indicada de repetição para resolvê-la é o `for`. Devido ao fato de que a soma está sempre incrementando o numerador e o denominador, então o `for` é perfeito para isso. Precisamos somente definir a regra da formação do numerador e do denominador em cada laço. Na própria definição do problema já nos foi dado o formato da atualização do numerador (variável `numerador`) e denominador (variável `denominador`): o numerador soma de 1 em 1, e o denominador depende do numerador considerado no laço. Em cada laço iremos adicionando o valor do cálculo da fração à soma (variável `soma`). Confira tudo isso no Código 6.9.

■ **Código 6.9** Solução do Problema 6.6.3 com `for`[10].

```
1 var n = parseInt(prompt("Informe o número n, inteiro e positivo:"));
2 var soma = 0;
3 for (var i=1,numerador=1,denominador=1;i<=n;i++,numerador++,denominador
    =2*numerador-1){
4   soma += numerador/denominador;
5 }
6 console.log("Soma =",soma);
```

■

Perceberam algo de diferente no `for` do Código 6.9? Na parte da declaração coloquei a variável que controla o laço, `i`, e adicionei as variáveis `numerador` e `denominador` (separadas por vírgula). E, na seção de incremento, ou decremento, coloquei a atualização de todas essas três variáveis. Perceba que o incremento não se aplica somente aos operadores `++` e `--`, você pode colocar outra forma de incremento como a que coloquei neste exemplo, `denominador=2*numerador-1`.

6.7 Resumo

Vimos três modos de repetir um fluxo em um algoritmo e utilizando *javascript*, um resumo dos comandos podem ser observados na Tabela 6.1. Com a prática você irá

[10] <https://replit.com/@DaniloBorges/EstudoCaso-Ex3-ComFor>

percebendo qual estrutura utilizar, em alguns casos uma estrutura será melhor que outra, e em outros casos podemos usar qualquer uma das estruturas. Agora é a sua vez, pratique bastante!

Comando	Descrição
`while(<condição>) {<instruções>}`	Enquanto a condição for satisfeita (verdade) as instruções serão executadas.
`do{<instruções>} while(<condição>);`	Nessa estrutura as instruções serão executadas pelo menos uma vez, após avaliada a condição o bloco do `do` repete caso a condição seja verdade.
`for(<declaração>;<condição>;<incremento>) {<instruções>}`	Declaração: será a inicialização das variáveis que serão incrementadas ou decrementadas no final do laço; Condição: é a avaliação que será feita para que o laço possa ser executado; e, Incremento: será inserida a operação de incremento, ou decremento, da variável declarada.
`break`	Para o fluxo da repetição.

Tabela 6.1: Resumo dos comandos das estruturas de repetição em *javascript*.

6.8 Exercícios

Exercício 6.1 Faça um programa que imprime a tabuada da soma, dado um número n. Exemplo: caso $n = 4$, seu programa deve imprimir:

```
4 + 0 = 4
4 + 1 = 5
4 + 2 = 6
4 + 3 = 7
4 + 4 = 8
4 + 5 = 9
4 + 6 = 10
4 + 7 = 11
4 + 8 = 12
4 + 9 = 13
4 + 10 = 14
```

■

Exercício 6.2 Faça um programa que imprime a tabuada da divisão, dado um número n. Exemplo: caso $n = 3$, seu programa deve imprimir:

```
3 / 3 = 1
6 / 3 = 2
9 / 3 = 3
```

```
12 / 3 = 4
15 / 3 = 5
18 / 3 = 6
21 / 3 = 7
24 / 3 = 8
27 / 3 = 9
30 / 3 = 10
```

Exercício 6.3 Faça um programa que calcula a seguinte soma:

$$\frac{1}{1}+\frac{3}{2}+\frac{5}{3}+\frac{7}{4}+\cdots+\frac{99}{50}$$

Exercício 6.4 Faça um programa que escreve os números ímpares entre 1 e 100.

Exercício 6.5 Faça um programa lê cinco números e exibe o maior e o menor número.

Exercício 6.6 Faça um programa que leia um número, n, e imprima se ele é primo ou não. (um número primo tem apenas 2 divisores: 1 e ele mesmo! O número 1 não é primo!!!)

Exercício 6.7 Escreva um programa que recebe um número entre 1 e 1000. Entre todos os números de 1 até o número informado você deve exibir aqueles que divididos por 11 dão resto 5.

Exercício 6.8 Faça um programa que leia um número, n, (informado pelo usuário) e mostre na tela os n primeiros números primos. Exemplo.: se for inserido o número 6, deverá ser impresso os números: 2, 3, 5, 7, 11 e 13.

Exercício 6.9 Escreva um programa que calcule e mostre a média aritmética dos números lidos entre 13 e 73.

Exercício 6.10 Escreva um programa que gera e escreve os 5 primeiros números perfeitos. Um número perfeito é aquele que é igual a soma dos seus divisores. (Ex.: $6 = 1+2+3$; $28 = 1+2+4+7+14$).

Exercício 6.11 Faça um algoritmo que calcule o fatorial de um número. (Ex.: $5! = 5\times4\times3\times2\times1 = 120$).

Exercício 6.12 Faça um programa que leia uma quantidade não determinada de números positivos. Calcule a quantidade de números pares e ímpares, a média de valores pares e a média geral dos números lidos. O número que encerrará a leitura será um número menor ou igual a zero.

Exercício 6.13 Faça um programa que leia uma quantidade desconhecida de números e conte quantos deles estão nos seguintes intervalos: [0-25], [26-50], [51-75] e [76-100]. Você deve parar de ler os números caso o número informado esteja fora do intervalo (0 a 100). ■

Exercício 6.14 Faça um programa que lê um valor n inteiro e positivo e que calcula e escreve o valor de E.

$$E = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ... + \frac{1}{(n-1)!} + \frac{1}{n!}$$

Observação: $0! = 1$. ■

7. Estrutura de Função

Neste capítulo você aprenderá a:

- Reaproveitar o código;
- Trabalhar com estrutura de funções no *javascript*.

7.1 Reaproveitamento de Fluxo

Nossa, olha só onde você chegou! No último capítulo. Eu estou feliz por você ter chegado até aqui, esse capítulo fecha este livro com "chave de ouro"! Vamos aprender agora como poderemos reaproveitar um código, transformando-o em uma função. Uma função na programação é como uma função na matemática, lembra? Talvez não, né..., mas, vamos relembrar um pouco para que consigamos fazer uma conexão aqui.

Suponhamos que temos a seguinte função: $f(x,y) = 8x - y + 20$. Se acalme, não é para resolver! Essa função podemos dizer que é constituída de duas partes: os parâmetros, x e y, e a resposta, $8x - y + 20$. Se eu lhe informar o valor de x e y você saberia a resposta, certo? Vamos tentar, se $x = 3$ e $y = 10$ qual o valor da resposta? Resposta correta: $8 \times 3 - 10 + 20 = 34$. De forma similar faremos na programação. Definiremos os parâmetros que serão incorporados no escopo da função e, se quisermos, definiremos um retorno que poderá ser utilizado em outra parte do programa.

Vamos partir para uma ideia de utilização mais prática do reaproveitamento do fluxo. Digamos que queiramos calcular a seguinte função:

$$C(n,p) = \frac{n!}{(n-p)!p!}. \tag{7.1}$$

Onde, a exclamação indica o sinal de fatorial; exemplo: $4! = 4 \times 3 \times 2 \times 1 = 24$. Se tivéssemos uma estrutura que calcula o fatorial nossa vida seria bem mais fácil. Só teríamos o trabalho de *invocá-la*. Daqui a pouco te explico como vai ser isso.

7.2 Função

Na determinação de uma função precisamos de duas coisas, os parâmetros e o retorno. Mas, além disso, podemos colocar o que quisermos em nossas funções. Usualmente, o que queremos retornar tem a ver com o parâmetro. Na Equação: 7.1, vemos isso nitidamente, o retorno depende dos valores de n e p. Porém pode ocorrer de uma função não possuir retorno, neste caso a chamamos de procedimento. Quando uma função é chamada está associado eu gosto de utilizar o termo invocação, ao invocá-la o programa irá aguardar o retorno da função.

Podemos, também definir funções sem parâmetros, elas funcionariam como funções constantes, onde não importa os valores que passarmos o retorno sempre será igual. Por exemplo: $f(x) = 5$; não importa o valor de x o retorno será sempre 5, nesses casos não precisamos colocar parâmetros, concorda? Bem, é isso que quero que você mentalize, por hora, a respeito de funções:

1. Funções podem possuir parâmetros ou não;
2. Funções que não possuem retorno são chamadas de procedimento;
3. Quando uma função é invocada o programa aguarda o seu retorno para continuar.

7.3 Função no Javascript

No *javascript* primeiramente declaramos uma função, e só depois a invocamos, podemos declará-la com a seguinte estrutura:

```
1 function <identificador> (<parâmetro(s)>) {
2     <instruções>
3     return <dado>;
4 }
```

Primeiramente você precisará definir um `identificador` para função, a definição do `identificador` deve seguir as mesmas regras aplicadas na definição de variáveis que falamos na Seção 3.4. Logo após, na `function`, você pode colocar quantos parâmetros quiser, separados por vírgula, (ou não colocar nenhum), e entre os parênteses.

> Na ausência de parâmetros você deve manter os parênteses, caso contrário será acusado um erro. Os parâmetros, quando existirem, devem seguir as mesmas regras da definição de variáveis.

É como se fossem as variáveis que ficam na parte de declaração do `for`, porém aqui você não precisará colocar o `var`, mas, se quiser, poderá inicializá-las. Em instruções, você pode colocar os comandos que quiser. E, opcionalmente, você pode colocar um `return`. Após o `return` você pode especificar o dado que você quer retornar. Pode ser, por exemplo, um `number`, um `boolean` ou uma `string`. Esse comando tem um poder similar ao do `break` em uma repetição. Tudo que estiver abaixo desse comando não será processado na função. Vamos a um exemplo prático...

No Código 7.1, temos a declaração da função `fatorial` para o cálculo do fatorial de um número `n`.

Código 7.1 Função Fatorial[1].

```
1 function fatorial(n){
2   var fat = 1;
3   for(var i=n;i>1;i--){
4     fat *= i;
```

[1] <https://replit.com/@DaniloBorges/FuncaoFatorial>

```
5     }
6     return fat;
7   }
8
9   var n = parseInt(prompt("Informe o valor de n"));
10
11  console.log(`${n}! = ${fatorial(n)}`);
```

■

A função `fatorial` calcula o fatorial do número usando o comando de repetição e retorna esse valor (linha 6). O parâmetro necessário é o número: `n`, que se deseja calcular o fatorial. A invocação da função é feita na linha 11. Observe que a declaração de uma função não implicará em sua execução. A mesma só será executada quando invocada.

Você só poderá invocar uma função após tê-la declarado. E, poderá invocar quantas vezes quiser.

Vamos agora criar uma função que calcula a Equação 7.1, no Código 7.2.

■ **Código 7.2** Função Combinação[2].

```
1   function fatorial(n){
2     var fat = 1;
3     for(var i=n;i>1;i--){
4       fat *= i;
5     }
6     return fat;
7   }
8
9   function combinacao(n,p){
10    return fatorial(n)/(fatorial(n-p)*fatorial(p));
11  }
12
13  var n = parseInt(prompt("Informe o valor de n"));
14  var p = parseInt(prompt("Informe o valor de p"));
15
16  console.log(`C(${n},${p}) = ${combinacao(n,p)}`);
```

■

Nesse código vemos o tal do reaproveitamento de forma mais evidente. Estamos reaproveitando a função `fatorial` (linha 1) na função `combinacao` (linha 10). Obtemos do usuário os valores de `n` e `p` e devolvemos o valor calculado (linha 16). O processamento disso é simples. Na linha 11, quando é realizada a primeira invocação ao fatorial, `fatorial(n)`, o programa executa a função e espera o retorno, e depois processa as demais invocações, `fatorial(n-p)` e `fatorial(p)`, após isso a operação de multiplicação e divisão é realizada com os valores retornados.

Oriento que você trabalhe aos poucos com a ideia de funções, pois, no *javascript* um dos tipos de dados que falei é `function` e, com ela podemos fazer muitas manipulações, mas não falarei disso aqui. Fica para uma próxima oportunidade.

Com isso eu encerro esse o assunto deste livro, espero que tenha gostado do conteúdo e não fique por aqui, seja curioso. O primeiro passo foi dado, continue a sua jornada na programação!

[2] <https://replit.com/@DaniloBorges/FuncaoCombinacao>

7.4 Exercícios

Exercício 7.1 Faça uma função que recebe um valor e verifica se o valor é positivo ou negativo. A função deve retornar um valor booleano (`true` para positivo e `false` para negativo). ■

Exercício 7.2 Faça uma função que recebe um valor inteiro e verifica se o valor é par ou ímpar. A função deve retornar um valor booleano (`true` para par e `false` para ímpar). ■

Exercício 7.3 Escreva uma função que recebe por parâmetro um valor inteiro e positivo n e retorna o valor de S. Para $n \geq 1$:

$$S = \frac{2}{4} + \frac{5}{5} + \frac{10}{6} + \frac{17}{7} + \frac{26}{8} + \cdots + \frac{(n^2+1)}{(n+3)}$$

. ■

Exercício 7.4 Faça um programa que calcula o valor de função de primeiro grau $f(x)$:

$$f(x) = ax + b$$

o usuário irá informar os valores de x, a e b. ■

Exercício 7.5 Faça um programa que calcule o montante final de uma aplicação com juros simples baseado na seguinte função:

$$M(C,i,t) = C \times (1 + i \times t)$$

onde, C é o capital inicial, i é a taxa de juros, valor entre 0 e 1, e t é a quantidade de meses. Use uma função. Os parâmetros, C, i e t serão informados pelo usuário. ■

Exercício 7.6 Faça um programa que calcule o montante final de uma aplicação com juros compostos baseado na seguinte função:

$$M(C,i,t) = C \times (1 + i)^t.$$

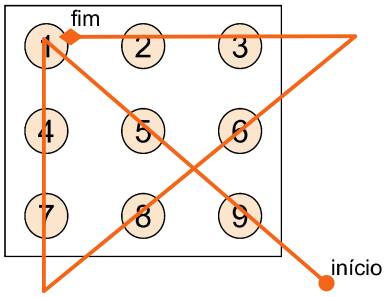
Onde, C é o capital inicial, i é a taxa de juros, valor entre 0 e 1, e t é a quantidade de meses. Use uma função. Os parâmetros, C, i e t serão informados pelo usuário. ■

Exercícios Resolvidos

1.3 Capítulo 1

1.1. a) e d).

1.3. Você deve considerar que pode sair do espaço limitado, pois não existe premissa que diz que não pode sair. Entre as várias possibilidades temos a seguinte:



2.4 Capítulo 2

2.1. d).

2.3. b).

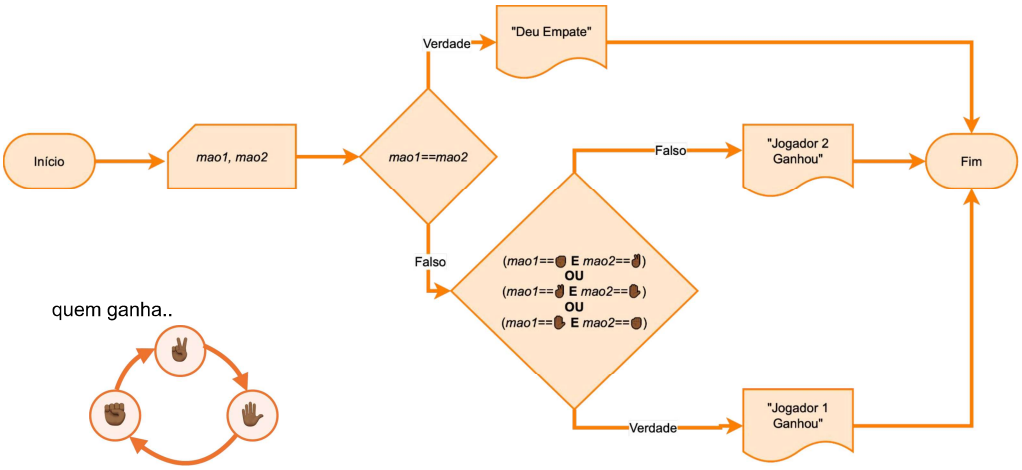
2.5. Representação narrativa:

1 - Obter número, n.

2 - Se $n >= 0$, então imprimir que o número n é positivo.

3 - Senão, imprimir que o número n é negativo.

2.7. Representação com fluxograma:



3.7 Capítulo 3

3.1. a), b), d) e f).
3.3. d).
3.5. Válidos: a), c) e e). Inválidos: b) por inicializar com um número; d) por conter um caractere especial, @.

4.4 Capítulo 4

4.1. a), c) e d).
4.3. b), c), d) e e).
4.5. Código do produto de dois números[3]:

```
1 var operando1 = Number(prompt("Digite o primeiro número: "));
2 var operando2 = Number(prompt("Digite o segundo número: "));
3 var resultado = operando1 * operando2;
4 console.log(`${operando1} x ${operando2} = ${resultado}`);
```

5.7 Capítulo 5

5.1. Código A+B>C[4]:

```
1 var A = Number(prompt("Informe o valor de A"));
2 var B = Number(prompt("Informe o valor de B"));
3 var C = Number(prompt("Informe o valor de C"));
4 if((A+B)<C){
5   console.log(`${A}+${B} é menor que ${C}`);
6 }else{
7   console.log(`${A}+${B} é maior ou igual a ${C}`);
8 }
```

[3] <https://replit.com/@DaniloBorges/45>
[4] <https://replit.com/@DaniloBorges/51>

5.3. Código pega dados[5]:

```
var nome = prompt("Informe seu nome");
var sexo = parseInt(prompt("Informe o seu sexo, 0 para masculino,
    1 para feminino"));
var casado = 0;
if(sexo==1){
  casado = parseInt(prompt("Coloque 1 se for casada, senão coloque
      0"));
}else{
  casado = parseInt(prompt("Coloque 1 se for casado, senão coloque
      0"));
}
var tempo = 0;
if(casado){
  if(sexo==1){
    tempo = parseInt(prompt("A quanto tempo é casada"));
  }else{
    tempo = parseInt(prompt("A quanto tempo é casado"));
  }
}
```

5.5. Código soma diferente caso par ou ímpar[6]:

```
var numero = parseInt(prompt("Informe um número"));
if(numero%2==0){
  console.log(`${numero} é par, logo o resultado da soma é ${
      numero+5}`);
}else{
  console.log(`${numero} é par, logo o resultado da soma é ${
      numero+8}`)
}
```

5.7. Código perfil eleitoral[7]:

```
var idade = parseInt(prompt("Qual a sua idade?"));
if(idade<16){
  console.log("Não é eleitor.");
}else if (idade>=18 && idade<65){
  console.log("Eleitor obrigatório.");
}else{
  console.log("Eleitor facultativo.")
}
```

5.9. Código forma de pagamento[8]:

```
console.log("=== Programa Forma de Pagamento ===");
var valor = Number(prompt("Informe o valor a ser pago"));
console.log("Selecione uma das formas de pagamento");
console.log("[1] para pagamento à vista em dinheiro ou cheque (
    desconto 10%)");
console.log("[2] para pagamento à vista no cartão de crédito (
    desconto 15%)");
console.log("[3] para duas vezes, sem juros");
console.log("[4] para duas vezes, com juros (10%)");
var opcao = parseInt(prompt("Opção"));
switch(opcao){
  case 1:{
```

[5] <https://replit.com/@DaniloBorges/53>
[6] <https://replit.com/@DaniloBorges/55>
[7] <https://replit.com/@DaniloBorges/57>
[8] <https://replit.com/@DaniloBorges/59>

```
11      console.log(`Valor de etiqueta: R$ ${valor}`);
12      console.log(`Valor a ser pago: R$ ${valor*0.9}`);
13      break;
14    }
15    case 2:{
16      console.log(`Valor de etiqueta: R$ ${valor}`);
17      console.log(`Valor a ser pago: R$ ${valor*0.85}`);
18      break;
19    }
20    case 3:{
21      console.log(`Valor de etiqueta: R$ ${valor}`);
22      console.log(`Valor a ser pago: 2x de R$ ${valor/2}`);
23      break;
24    }
25    case 4:{
26      console.log(`Valor de etiqueta: R$ ${valor}`);
27      console.log(`Valor a ser pago: 2x de R$ ${valor*1.1/2}`);
28      break;
29    }
30  }
```

5.11. Código informa mês[9]:

```
1  var mes = parseInt(prompt("Informe um número entre 1 e 12"));
2  switch(mes){
3    case 1:{
4      console.log("Janeiro.");
5      break;
6    }
7    case 2:{
8      console.log("Fevereiro.");
9      break;
10   }
11   case 3:{
12     console.log("Março.");
13     break;
14   }
15   case 4:{
16     console.log("Abril.");
17     break;
18   }
19   case 5:{
20     console.log("Maio.");
21     break;
22   }
23   case 6:{
24     console.log("Junho.");
25     break;
26   }
27   case 7:{
28     console.log("Julho.");
29     break;
30   }
31   case 8:{
32     console.log("Agosto.");
33     break;
34   }
35   case 9:{
36     console.log("Setembro.");
37     break;
```

[9]<https://replit.com/@DaniloBorges/511>

```
38    }
39    case 10:{
40      console.log("Outubro.");
41      break;
42    }
43    case 11:{
44      console.log("Novembro.");
45      break;
46    }
47    case 12:{
48      console.log("Dezembro.");
49      break;
50    }
51    default:{
52      console.log("Não existe mês para opção informada.");
53    }
54  }
```

6.8 Capítulo 6

6.1. Código tabuada da soma[10]:

```
1  console.log("=== Tabuada de Soma ===");
2  var numero = Number(prompt("Informe qual tabuada de soma você quer
       "));
3  for (var operador=0;operador<=10;operador++) {
4    soma = numero + operador;
5    console.log(`${numero} + ${operador} = ${soma}`);
6  }
```

6.3. Código somatório[11]:

```
1  console.log("=== Somatório ===");
2  console.log("Exibe o somatório de 1/1+3/2+5/3+...+99/50");
3  var somatorio = 0;
4  for (var operador=0;operador<50;operador++) {
5    somatorio += (2*operador+1)/(operador+1);
6  }
7  console.log(`Resultado: ${somatorio}`);
```

6.5. Código pega maior e menor[12]:

```
1  console.log("=== Maior e Menor ===");
2  var maior = -Infinity, menor = Infinity;
3  var contador = 1;
4  do{
5    var numero = Number(prompt(`Informe o ${contador}° número`));
6    if(numero>maior){
7      maior = numero;
8    }
9    if(numero<menor){
10     menor = numero;
11   }
12   contador++;
13 }while(contador<=5);
```

[10] <https://replit.com/@DaniloBorges/61>
[11] <https://replit.com/@DaniloBorges/63>
[12] <https://replit.com/@DaniloBorges/65>

```
14
15 console.log(`O maior número é: ${maior}`);
16 console.log(`O menor número é: ${menor}`);
```

6.7. Código números módulo 11 resto 5[13]:

```
1  console.log("=== Números%11 = 5  ===");
2  var numero = Number(prompt(`Informe um número entre 1 e 1000`));
3  if (!(numero>=1 && numero<=1000)){
4    console.log("Número informado não está no intervalo definido
       entre 1 e 1000.")
5  }else{
6    var contador = 0;
7    var incremento = 1;
8    while (incremento<=numero){
9      if(incremento%11==5){
10       contador++;
11       console.log(`${contador}° encontrado: ${incremento}`);
12     }
13     incremento++;
14   }
15   if(contador==0){
16     console.log("Nenhum número encontrado.");
17   }else{
18     console.log(`Foram encontrado(s) ${contador} número(s).`);
19   }
20 }
```

6.11. Código calcula fatorial[14]:

```
1  console.log("=== Cálculo do Fatorial  ===");
2  var numero = parseInt(prompt("Informe um número inteiro e positivo
     "));
3  if (numero<0){
4    console.log(`Não é possível calcular o fatorial deste número: ${
       numero}.`);
5  }else{
6    var fatorial = 1;
7    for(var i=numero;i>1;i--){
8      fatorial *= i;
9    }
10   console.log(`${numero}! = ${fatorial}.`);
11 }
```

6.13. Código quantidade por intervalo[15]:

```
1  console.log("=== Quantidade no Intervalo ===");
2  var total = 0;
3  var contador_de_0_25 = 0;
4  var contador_de_26_50 = 0;
5  var contador_de_51_75 = 0;
6  var contador_de_76_100 = 0;
7  do{
8    var numero = Number(prompt(`Informe o ${total+1}° número`));
9    if(numero>100 || numero<0){
10     console.log(`Você digitou um número fora do intervalo`);
11     break;
12   }
```

[13] <https://replit.com/@DaniloBorges/67>
[14] <https://replit.com/@DaniloBorges/611>
[15] <https://replit.com/@DaniloBorges/613>

```
13    total++;
14    if(numero>=76){
15      contador_de_76_100++;
16    }else if(numero>=51){
17      contador_de_51_75++;
18    }else if(numero>=26){
19      contador_de_26_50++;
20    }else{
21      contador_de_0_25++;
22    }
23  }while(true);
24  console.log(`Foram informados ${total} número(s).`)
25  //o \t cria um tab no texto
26  console.log(`\tDe [0,25]: ${contador_de_0_25}`);
27  console.log(`\tDe [26,50]: ${contador_de_26_50}`);
28  console.log(`\tDe [51,75]: ${contador_de_51_75}`);
29  console.log(`\tDe [76,100]: ${contador_de_76_100}`);
```

7.4 Capítulo 7

7.1. Código verifica positivo[16]:

```
1  console.log("=== Verifica se Positivo ===")
2  function eh_positivo(numero){
3    return (numero>=0);
4  }
5  var numero = Number(prompt("Informe um número"));
6  console.log(`${numero} é positivo? ${eh_positivo(numero)}`);
```

7.3. Código calcula S[17]:

```
1  console.log("=== Calcula S ===")
2  function calcula_s(numero){
3    if(numero<0){
4      return 0;
5    }else{
6      var somatorio = 0;
7      for(var n=1;n<=numero;n++){
8        somatorio += (n**2+1)/(n+3);
9      }
10     return somatorio;
11   }
12 }
13 var numero = Number(prompt("Informe um número"));
14 console.log(`O somatório de ${numero} termo(s) é: ${calcula_s(
       numero)}`);
```

7.5. Código juros simples[18]:

```
1  console.log("=== Juros Simples ===")
2  function montante_juros_simples(capital,taxa,tempo){
3    return capital*(1+taxa*tempo);
4  }
5  var C = Number(prompt("Informe o capital inicial em R$"));
6  var i = Number(prompt("Informe taxa de juros entre 0 e 100 (%)"));
```

[16] <https://replit.com/@DaniloBorges/71>
[17] <https://replit.com/@DaniloBorges/73>
[18] <https://replit.com/@DaniloBorges/75>

```
7  var t = Number(prompt("Informe tempo em meses"));
8  console.log(`O montante final será: R$ ${montante_juros_simples(C,
       i/100,t)}`);
```

Bibliografia

ALGORITMO e Lógica de Programação. Universidade Federal do Rio Grande do Norte, 2004.

ALMEIDA, M. E. B. de; VALENTE, J. A. Pensamento computacional nas políticas e nas práticas em alguns países. *Revista Observatório*, v. 5, n. 1, p. 202–242, 2019.

ALVES, G. F. de O. *Ebook Lógica de Programação para Iniciantes*. 2018.

FLANAGAN, D. *JavaScript: o guia definitivo*. [S.l.]: Bookman Editora, 2004.

JUNIOR, O. F. A. *Práticas inovadoras utilizando a informática como expediente de otimização e modernização do poder judiciário: a utilização da videoconferência*. Tese (Doutorado), 2009.

MARTINS, M. W. P. et al. Dificuldades de aprendizagem em lógica de programação dos alunos de tecnologia: Uma revisão bibliográfica. Instituto Federal Goiano, 2022.

MENEZES, N. N. C. Introdução a programação com python. *São Paulo: Novatec*, 2010.

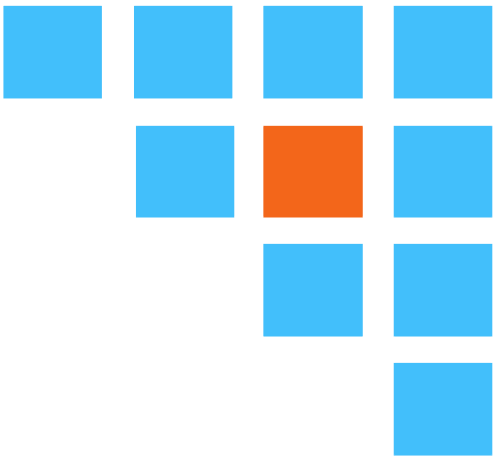
MORZE, N. V.; MASHKINA, I. V.; BOIKO, M. A. Experience in training specialists with mathematical computer modeling skills, taking into account the needs of the modern labor market. In: *CEUR Workshop Proceedings*. [S.l.: s.n.], 2022. p. 95–106.

SOUZA, F. A. de; FALCÃO, T. P.; MELLO, R. F. O ensino de programação na educação básica: Uma revisão da literatura. In: SBC. *Anais do XXXII Simpósio Brasileiro de Informática na Educação*. [S.l.], 2021. p. 1265–1275.

TILKOV, S.; VINOSKI, S. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, IEEE, v. 14, n. 6, p. 80–83, 2010.

VASCONCELLOS, I. L. B.; TAMARIZ, A. D. R.; BATISTA, S. C. F. *Apostila de Lógica de Programação*. Instituto Federal Fluminense - Campus Campos Centro, 2019.

VIEIRA, A. R. L. et al. A lógica no contexto da pesquisa científica: Uma abordagem metodológica. *Revista Ibero-Americana de Humanidades, Ciências e Educação*, v. 8, n. 1, p. 1730–1746, 2022.



Programação para Todos

Do ZERO ao Básico

Editora da
Universidade
Estadual do Piauí

editora.uespi.br