



INSTITUTO FEDERAL

Espírito Santo

Campus Cachoeiro de Itapemirim

Lógica de Programação com JavaScript

Apostila Didática

Prof. Rafael Vargas Mesquita

CURSO REPROGRAME-SE, INSTITUTO FEDERAL DO ESPÍRITO SANTO - CAMPUS CACHOEIRO

Caro estudante,

Bem-vindo à apostila de Lógica de Programação com JavaScript! Este material foi desenvolvido para proporcionar a você uma introdução sólida aos conceitos fundamentais da lógica de programação, bem como uma compreensão prática de como aplicar esses conceitos utilizando a linguagem de programação JavaScript.

A lógica de programação é o alicerce de todo desenvolvimento de software. Ela nos permite estruturar e organizar nossos pensamentos, transformando problemas complexos em soluções claras e eficientes. Ao dominar os conceitos apresentados nesta apostila, você estará preparado para enfrentar desafios de programação com confiança.

Ao longo deste material, você encontrará explicações detalhadas, exemplos de código e exercícios práticos que o guiarão no processo de aprendizado. Começaremos com uma introdução à linguagem JavaScript, abordando sua sintaxe, variáveis, tipos de dados e operadores. Em seguida, mergulharemos nos comandos de decisão, permitindo que seu código tome decisões lógicas com base em diferentes condições.

Continuando, exploraremos os comandos de repetição, que são fundamentais para a execução de tarefas repetitivas em um programa. Você aprenderá a utilizar laços de repetição como o **for**, **while** e **do-while**, bem como a aplicar instruções de controle como **break** e **continue**.

Apostila de Lógica de Programação com JavaScript também oferece uma introdução às coleções de dados por meio dos arrays, explorando a criação, manipulação e iteração dessas estruturas. Você aprenderá sobre os principais métodos disponíveis para trabalhar com arrays, como **push**, **pop**, **shift**, **unshift**, **splice**, **sort** e **indexOf**.

Por fim, abordaremos as funções e procedimentos, que são blocos de código reutilizáveis e modulares. Você aprenderá a definir e chamar funções, trabalhar com argumentos padrão, entender o escopo de variáveis, utilizar expressões de função e aproveitar as vantagens das Arrow Functions.

Esteja preparado para mergulhar no mundo da lógica de programação e expandir seus conhecimentos em JavaScript. Ao final desta apostila, você estará equipado com as habilidades necessárias para resolver problemas, criar algoritmos eficientes e escrever programas robustos utilizando a linguagem JavaScript.

Desejamos a você uma jornada de aprendizado estimulante e proveitosa. Que esta apostila seja sua aliada na busca por um domínio sólido da lógica de programação com JavaScript. Pronto para embarcar nessa jornada? Vamos começar!

Copyright © 2023 Rafael Vargas Mesquita

INSTITUTO FEDERAL DO ESPÍRITO SANTO CAMPUS CACHOEIRO

1ª Ed., Julho de 2023

Sumário

I

Parte I: Conceitos

1	Introdução a Linguagem JavaScript	17
1.1	Histórico da Linguagem	17
1.2	Sintaxe	18
1.2.1	Palavras-chave	18
1.2.2	Instruções	18
1.2.3	Blocos de Código	18
1.2.4	Comentários	19
1.3	Variáveis e Tipos de Dados	19
1.3.1	Declaração de Variáveis	19
1.3.2	Atribuição de Valores	21
1.3.3	Tipos de Dados	22
1.3.4	Mostrando o conteúdo de Variáveis	22
1.4	Operadores	22
1.4.1	Operadores aritméticos	22
1.4.2	Operadores relacionais	23
1.4.3	Operadores lógicos	24
1.4.4	Operadores de atribuição	24

1.5	Entrada de Dados	25
1.5.1	Entrada de dados utilizando o módulo prompt-sync	25
1.5.2	Conversão de tipos de dados	25
1.6	Exercícios Propostos	26
1.7	Gabarito dos Exercícios	26
2	Comandos de Decisão	27
2.1	Operador Ternário	27
2.2	Estrutura if-else	28
2.3	Estrutura if-elseif-else	29
2.4	Comandos de Decisão Aninhados	31
2.5	Exercícios Propostos	34
2.6	Gabarito dos Exercícios	35
3	Comandos de Repetição	37
3.1	Laço for	38
3.2	Laço while	39
3.3	Laço do ... while	40
3.4	Controle de laço: break e continue	41
3.5	Comandos de Repetição Aninhados	43
3.6	Exercícios Propostos	46
3.7	Gabarito dos Exercícios	47
4	Coleções de dados (Arrays)	49
4.1	Criando um Array	49
4.2	Referenciando um Array	50
4.3	Iterando um Array	50

4.4	Métodos de um Array	51
4.5	Exercícios Propostos	54
4.6	Gabarito dos Exercícios	54
5	Funções e Procedimentos	55
5.1	Definição de Função	56
5.2	Chamada de Função	56
5.3	Argumentos Default	57
5.4	Escopo de Variáveis	57
5.5	Expressão de Função	58
5.6	Arrow Functions	58
5.7	Comparativo entre os tipos de funções	59
5.8	Exercícios Propostos	60
5.9	Gabarito dos Exercícios	60

II

Parte II: Práticas

6	Práticas Propostas	63
6.1	Operadores	64
6.1.1	Prática Proposta	64
6.1.2	Gabarito da Prática	64
6.2	Entrada de Dados	65
6.2.1	Prática Proposta	65
6.2.2	Gabarito da Prática	65
6.3	Comandos de Decisão	66
6.3.1	Prática Proposta	66
6.3.2	Gabarito da Prática	66

6.4	Comandos de Decisão Aninhados	67
6.4.1	Prática Proposta	67
6.4.2	Gabarito da Prática	67
6.5	Comandos de Repetição	69
6.5.1	Prática Proposta	69
6.5.2	Gabarito da Prática	69
6.6	Comandos de Repetição Aninhados	71
6.6.1	Prática Proposta	71
6.6.2	Gabarito da Prática	71
6.7	Coleção de Dados (Arrays)	73
6.7.1	Prática Proposta	73
6.7.2	Gabarito da Prática	73
6.8	Funções e Procedimentos	76
6.8.1	Prática Proposta	76
6.8.2	Gabarito da Prática	76

Lista de Figuras

1.1	Histórico da Linguagem	17
2.1	Diagrama de blocos sobre a estrutura if-else.	29
2.2	Diagrama de blocos sobre a estrutura if-elseif-else	31
2.3	Diagrama de blocos sobre comandos de decisão aninhados	33
3.1	Comparativo entre os comandos de repetição	41
3.2	Diagrama de blocos sobre comandos de repetição aninhados	45
4.1	Visualização de Arrays.	50
4.2	Visualização dos métodos de Arrays.	51
6.1	Problema do cálculo do salário líquido de um cidadão.	63

Lista de Exemplos de Códigos

1.1	Bloco de Código	18
1.2	Comentários	19
1.3	Declaração de variável com var	20
1.4	Declaração de variável com let	20
1.5	Declaração de variável com const	21
1.6	Atribuição de valores	21
1.7	Mostrando conteúdo de variáveis	22
1.8	Operadores aritméticos	23
1.9	Operadores relacionais	23
1.10	Operadores lógicos	24
1.11	Operadores de atribuição	24
1.12	Entrada de dados com prompt-sync	25
1.13	Conversão de tipos de dados	25
2.1	Sintaxe do operador ternário	27
2.2	Operadores ternário	27
2.3	Sintaxe da estrutura if-else	28
2.4	Estrutura if-else	28
2.5	Sintaxe da estrutura if-elseif-else	30
2.6	Estrutura if-elseif-else	30
2.7	Comandos de decisão aninhados.	32
3.1	Sintaxe do laço for	38

3.2	Laço for	38
3.3	Sintaxe do laço while	39
3.4	Laço while	39
3.5	Sintaxe do laço do ... while	40
3.6	Laço do ... while	40
3.7	Controle de laço com break	42
3.8	Controle de laço com continue	42
3.9	Sintaxe dos comandos de repetição aninhados	43
3.10	Comandos de repetição aninhados	43
4.1	Criando um array	49
4.2	Referenciando um array	50
4.3	Iterando um array com for	50
4.4	Iterando um array com for...of	51
4.5	Método push de um array	52
4.6	Método pop de um array	52
4.7	Método shift de um array	52
4.8	Método unshift de um array	52
4.9	Método splice de um array	52
4.10	Método sort de um array	53
4.11	Método indexOf de um array	53
5.1	Definição de função	56
5.2	Chamada de função	56
5.3	Argumentos Default	57
5.4	Escopo de variáveis de função	57
5.5	Expressão de função	58
5.6	Arrow functions	58
5.7	Comparativo entre os tipos de funções	59
6.1	Prática - Operadores	64
6.2	Prática - Entrada de Dados	65

6.3	Prática - Comandos de Decisão	66
6.4	Prática - Comandos de Decisão Aninhados	68
6.5	Prática - Comandos de Repetição	70
6.6	Prática - Comandos de Repetição Aninhados	72
6.7	Prática - Coleção de Dados (Arrays)	75
6.8	Prática - Funções e Procedimentos	78



Parte I: Conceitos

1	Introdução a Linguagem JavaScript	17
2	Comandos de Decisão	27
3	Comandos de Repetição	37
4	Coleções de dados (Arrays)	49
5	Funções e Procedimentos	55

1. Introdução a Linguagem JavaScript

1.1 Histórico da Linguagem

JavaScript é uma linguagem de programação interpretada ¹, originalmente criada em 1995 por Brendan Eich enquanto trabalhava para a Netscape Communications Corporation. A linguagem foi criada para ser usada em conjunto com a plataforma Netscape Navigator, que era um dos navegadores mais populares da época. A linguagem foi originalmente chamada de Mocha, mas foi posteriormente renomeada para LiveScript e finalmente para JavaScript.



(a) Brendan Eich, nascido em 1961, criador da linguagem JavaScript.



(b) Foto de 2002 da sede da Netscape em Mountain View, Califórnia.

Figura 1.1: Histórico da Linguagem

O objetivo da linguagem JavaScript era permitir que os desenvolvedores web criassem interações mais dinâmicas e sofisticadas com os usuários, além de permitir que as páginas web fossem atualizadas dinamicamente, sem a necessidade de recarregar a página inteira. Desde a sua criação, JavaScript

¹É uma linguagem de programação em que o código fonte é executado por um programa de computador chamado interpretador, onde a interpretação e a execução do programa acontecem em tempo real.

evoluiu significativamente e hoje é uma das linguagens de programação mais populares do mundo, amplamente utilizada tanto no lado do cliente quanto do servidor.

1.2 Sintaxe

A sintaxe de uma linguagem de programação define as regras para escrever códigos naquela linguagem. Na linguagem JavaScript, a sintaxe inclui palavras-chave, instruções, blocos de código e comentários.

1.2.1 Palavras-chave

As palavras-chave são termos reservados que têm um significado específico na linguagem JavaScript. Por exemplo, **var** é uma palavra-chave que é usada para declarar variáveis. As palavras-chave são escritas em minúsculas e não podem ser usadas como nomes de variáveis, funções ou outras entidades no código.

1.2.2 Instruções

As instruções são comandos que executam uma ação em JavaScript. As instruções podem ser simples ou complexas e podem incluir operadores, variáveis, constantes e outras entidades. As instruções são separadas por ponto e vírgula.

1.2.3 Blocos de Código

Os blocos de código são uma coleção de instruções que são executadas juntas. Em JavaScript, os blocos de código são definidos usando chaves. Os blocos de código podem ser aninhados uns dentro dos outros e são frequentemente usados com instruções de decisão e repetição.

```
1 console.log("Mostrando de 1 até 10")
2 for (i = 1; i <= 10; i++) {
3     console.log("i = ", i);
4 }
5 console.log("O programa finaliza aqui!");
```

Exemplo de Código 1.1: Bloco de Código



Não se preocupe em entender as instruções desse trecho de código. O mais importante é saber que a instrução da linha 3 está dentro de um bloco de código.

1.2.4 Comentários

Os comentários são usados para adicionar notas explicativas ao código. Os comentários são ignorados pelo interpretador JavaScript e não afetam a execução do código. Em JavaScript, os comentários podem ser de uma linha (iniciados com `//`) ou de várias linhas (entre `/*` e `*/`).

```
1 // comentário de uma linha
2
3 /* isto é um comentário longo
4    de múltiplas linhas.
5    */
```

Exemplo de Código 1.2: Comentários

1.3 Variáveis e Tipos de Dados

Em JavaScript, as variáveis são usadas para armazenar dados temporários ou permanentes. As variáveis são declaradas usando as palavras-chave **var**, **let** ou **const** seguida pelo nome da variável. O valor da variável pode ser atribuído na mesma linha da declaração ou em uma linha posterior.

1.3.1 Declaração de Variáveis

Ao trabalhar com variáveis em JavaScript, temos três formas de declará-las: utilizando **var**, **let** e **const**. Embora todas sejam usadas para criar variáveis, cada uma possui características específicas. Nesta subseção, vamos explorar essas diferenças e entender quando utilizar cada uma delas.

var: A palavra-chave **var** era a forma tradicional de declarar variáveis em JavaScript antes da introdução do **let** e **const**. Uma variável declarada com **var** tem escopo de função ou escopo global, o que significa que ela é acessível dentro da função (ou bloco de código) em que foi declarada ou em todo o programa, caso tenha sido declarada fora de qualquer função (ou bloco de código). Além disso, **var** permite a redeclaração da mesma variável e permite que seu valor seja alterado. No entanto, o escopo de bloco não é respeitado pelo **var**, o que pode levar a comportamentos inesperados.

```
1 var nome = "Alice";
2
3 if (true) { // Início do bloco de código com comando de decisão (if)
4     var nome = "Bob";
5     console.log(nome); // Saída: Bob
6 }
7
8 console.log(nome); // Saída: Bob
```

Exemplo de Código 1.3: Declaração de variável com **var**

let: O **let** foi introduzido no ECMAScript 6 e resolve algumas limitações do **let**. Uma variável declarada com **let** tem escopo de bloco, o que significa que ela é acessível apenas dentro do bloco em que foi declarada. Além disso, o **let** não permite a redeclaração da mesma variável dentro do mesmo escopo e seu valor pode ser alterado normalmente.

```
1 let nome = "Alice";
2
3 if (true) { // Início do bloco de código com comando de decisão (if)
4     let nome = "Bob";
5     console.log(nome); // Saída: Bob
6 }
7
8 console.log(nome); // Saída: Alice
```

Exemplo de Código 1.4: Declaração de variável com **let**

const: A palavra-chave **const** é usada para declarar constantes, ou seja, variáveis cujo valor não pode ser alterado após a atribuição inicial. Assim como o **let**, o **const** tem escopo de bloco e não permite a reatribuição do valor da variável. É importante mencionar que, ao declarar uma constante, é necessário atribuir um valor a ela imediatamente.

```
1  const nome = "Alice";
2
3  if (true) { // Início do bloco de código com comando de decisão (if)
4      const nome = "Bob";
5      console.log(nome); // Saída: Bob
6  }
7
8  nome = "Carol"; // Erro: reatribuição de valor a uma constante (const)
```

Exemplo de Código 1.5: Declaração de variável com **const**



O comando de decisão **if** será explicado em seções posteriores. Está sendo utilizado nestes exemplos de declarações de variáveis apenas para especificar blocos de código.

É recomendado utilizar **let** para variáveis que precisam ser modificadas ao longo do programa, pois ele oferece um escopo de bloco mais seguro. Já o **const** é adequado para declarar variáveis cujo valor não deve ser alterado. O **var** ainda pode ser utilizado em contextos específicos, porém é recomendado evitar seu uso em favor do **let** e **const** devido às suas limitações e comportamentos inesperados.



Ao declarar variáveis em JavaScript, é importante entender as diferenças entre **var**, **let** e **const** e escolher a melhor opção de acordo com a necessidade do seu código. A escolha correta ajuda a garantir um código mais claro, conciso e menos propenso a erros.

1.3.2 Atribuição de Valores

Você pode atribuir valores a variáveis em JavaScript usando o operador de atribuição **"="**.

```
1  var nomeDaVariavel1 = "valor";
2  let nomeDaVariavel2 = "valor";
3  const nomeDaConstante1 = "valor";
```

Exemplo de Código 1.6: Atribuição de valores

1.3.3 Tipos de Dados

JavaScript tem seis tipos de dados primitivos: string, number, boolean, null, undefined e symbol. Além desses tipos de dados primitivos, o JavaScript também possui o tipo de dado objeto.

1.3.4 Mostrando o conteúdo de Variáveis

Para mostrar o conteúdo de uma variável, você pode usar a função **console.log()**. Por exemplo:

```
1 var nome = "Maria";  
2 console.log(nome); // Isso mostra "Maria" no console
```

Exemplo de Código 1.7: Mostrando conteúdo de variáveis

Em JavaScript, as variáveis são de tipagem dinâmica, o que significa que você não precisa declarar explicitamente o tipo de dado que uma variável contém. O tipo de dado é inferido automaticamente pelo JavaScript com base no valor atribuído à variável.

1.4 Operadores

Operadores são símbolos especiais que executam operações em valores ou variáveis. Em JavaScript, existem quatro tipos principais de operadores: aritméticos, relacionais, lógicos e de atribuição.

1.4.1 Operadores aritméticos

Os operadores aritméticos são usados para realizar operações matemáticas em valores numéricos. Aqui estão os principais operadores aritméticos em JavaScript:

- + (adição)
- - (subtração)
- * (multiplicação)
- / (divisão)
- % (módulo - retorna o resto da divisão)

```
1 let a = 5;  
2 let b = 2;  
3 console.log(a + b); // 7  
4 console.log(a - b); // 3  
5 console.log(a * b); // 10
```

```
6 console.log(a / b); // 2.5
7 console.log(a % b); // 1
```

Exemplo de Código 1.8: Operadores aritméticos**1.4.2 Operadores relacionais**

Os operadores relacionais são usados para comparar valores e retornar um valor booleano (true ou false). Aqui estão os principais operadores relacionais em JavaScript:

- > (maior que)
- < (menor que)
- >= (maior ou igual a)
- <= (menor ou igual a)
- == (igual a)
- != (diferente de)
- === (igualdade estrita - compara o valor e o tipo)
- !== (diferença estrita - compara o valor e o tipo)

```
1 let a = 5;
2 let b = 2;
3 console.log(a > b); // true
4 console.log(a < b); // false
5 console.log(a >= b); // true
6 console.log(a <= b); // false
7 console.log(a == b); // false
8 console.log(a != b); // true
9 console.log(a === "5"); // false
10 console.log(a !== "5"); // true
```

Exemplo de Código 1.9: Operadores relacionais

1.4.3 Operadores lógicos

Os operadores lógicos são usados para combinar ou negar valores booleanos. Aqui estão os principais operadores lógicos em JavaScript:

- **&&** (E lógico - retorna true se ambas as expressões são verdadeiras)
- **||** (OU lógico - retorna true se pelo menos uma das expressões é verdadeira)
- **!** (NÃO lógico - nega uma expressão booleana)

```
1 let a = true;
2 let b = false;
3 console.log(a && b); // false
4 console.log(a || b); // true
5 console.log(!a); // false
```

Exemplo de Código 1.10: Operadores lógicos

1.4.4 Operadores de atribuição

Os operadores de atribuição são usados para atribuir valores a variáveis. Aqui estão os principais operadores de atribuição em JavaScript:

- **=** (atribuição)
- **+=** (atribuição de adição)
- **-=** (atribuição de subtração)
- ***=** (atribuição de multiplicação)
- **/=** (atribuição de divisão)
- **%=** (atribuição de módulo)

```
1 let a = 5;
2 a += 2; // o mesmo que a = a + 2;
3 console.log(a); // 7
```

Exemplo de Código 1.11: Operadores de atribuição

1.5 Entrada de Dados

1.5.1 Entrada de dados utilizando o módulo prompt-sync

É possível utilizar o módulo prompt-sync para realizar a leitura de dados de entrada em um programa JavaScript. O módulo prompt-sync permite a leitura de dados de entrada de forma síncrona, ou seja, o programa aguarda a entrada de dados pelo usuário antes de prosseguir com a execução do código.

Para utilizar o módulo prompt-sync, devemos primeiro instalá-lo em nosso projeto utilizando o gerenciador de pacotes npm. Em seguida, podemos importar o módulo em nosso código e utilizá-lo para realizar a leitura de dados de entrada.

```
1 const prompt = require("prompt-sync")();
2 let nome = prompt("Qual é seu nome? ");
3 console.log("Oi, " + nome + "!");
```

Exemplo de Código 1.12: Entrada de dados com prompt-sync

Ao executar o código acima, a mensagem **Qual seu nome?** será exibida no console do terminal, e o usuário poderá digitar o seu nome. Em seguida, a mensagem **Oi, [nome]!** será exibida no console do terminal.

1.5.2 Conversão de tipos de dados

O valor retornado pelo módulo prompt-sync será sempre uma string. Se quisermos tratar esse valor como um número, por exemplo, devemos fazer a conversão de tipos de dados utilizando as funções parseInt() ou parseFloat().

```
1 const prompt = require("prompt-sync")();
2 let idade = parseInt(prompt("Qual é sua idade?"));
3 console.log("Você tem " + idade + " anos.");
```

Exemplo de Código 1.13: Conversão de tipos de dados

Nesse exemplo, o valor fornecido pelo usuário será convertido para um número inteiro utilizando a função parseInt(). Em seguida, a mensagem **Você tem [idade] anos.** será exibida no console do navegador.

1.6 Exercícios Propostos

Exercício 1.1 — Entrada de Dados: Exercício 01. Elabore um algoritmo para calcular o salário líquido de um determinado funcionário. Você deve receber os seguintes valores: salário e imposto de renda. Com base nestes valores você deverá encontrar o valor do salário líquido (salário líquido = salário – imposto de renda) e mostrar na tela o resultado.

Exercício 1.2 — Entrada de Dados: Exercício 02. Elabore um algoritmo para calcular o consumo de um determinado carro em um percurso qualquer. Você deve receber os seguintes valores: modelo do carro, número de quilômetros percorridos e número de litros de combustível gastos no percurso. Com base nestes valores você deverá encontrar o consumo (km/litro) do carro e mostrar na tela o resultado da seguinte forma:

O consumo do carro <modelo do carro> é de <consumo> km/litro.

1.7 Gabarito dos Exercícios

Gabarito 1.1 — Sintaxe e Tipos de Dados: Exercício 01.



Gabarito 1.2 — Entrada de Dados: Exercício 01.



Gabarito 1.3 — Entrada de Dados: Exercício 02.



2. Comandos de Decisão

Os comandos de decisão são utilizados em programação para avaliar uma condição e executar um conjunto de instruções específicas caso essa condição seja verdadeira. Essas estruturas são fundamentais para a lógica e o fluxo de controle em um programa. Em JavaScript, existem diversas formas de implementar comandos de decisão, cada uma com suas particularidades e casos de uso específicos. Ao utilizar comandos de decisão, podemos controlar o fluxo de execução de um programa, realizar verificações de validação, lidar com casos especiais e tomar decisões dinâmicas durante a execução. Compreender e utilizar corretamente os comandos de decisão é essencial para desenvolver programas funcionais e robustos. Eles nos ajudam a criar lógicas complexas, tornar nossos programas mais interativos e lidar com situações diversas de forma eficiente.

2.1 Operador Ternário

Para realizar testes em seu algoritmo é possível utilizar o operador ternário para implementar comandos de decisão em JavaScript. Esse operador permite avaliar uma condição e retornar um valor ou outro, dependendo do resultado da avaliação da condição. A sintaxe básica do operador ternário é a seguinte:

```
1 condicao ? valor_se_verdadeiro : valor_se_falso;
```

Exemplo de Código 2.1: Sintaxe do operador ternário

Aqui está um exemplo prático do uso do operador ternário:

```
1 const idade = 18;
2 const podeDirigir = idade >= 18 ? "Sim" : "Não";
3 console.log(podeDirigir); // Resultado: Sim
```

Exemplo de Código 2.2: Operadores ternário

Nesse exemplo, a variável `podeDirigir` recebe o valor **Sim** se a idade for maior ou igual a 18, caso contrário, recebe o valor **Não**. Neste caso, como a idade é 18, a condição é verdadeira e a variável `podeDirigir` recebe o valor **Sim**.

Podemos usar o operador ternário para tomar decisões simples e atribuir valores com base em uma condição, de forma concisa e legível. Ele é uma ótima opção quando a lógica é direta e não envolve ramificações complexas.

2.2 Estrutura if-else

No entanto, a estrutura mais comum para implementar um comando de decisão é a estrutura if-else. Essa estrutura permite avaliar uma condição e executar um conjunto de instruções caso a condição seja verdadeira, e outro conjunto de instruções caso a condição seja falsa. A sintaxe básica da estrutura if-else é a seguinte:

```
1 if (condição) {
2     // instruções executadas caso a condição seja verdadeira
3 } else {
4     // instruções executadas caso a condição seja falsa
5 }
```

Exemplo de Código 2.3: Sintaxe da estrutura if-else

A fim de ilustrar o uso do comando if-else, considere o seguinte exemplo:

```
1 const idade = parseInt(prompt("Digite a idade:"));
2 if (idade >= 18) {
3     console.log("Você é maior de idade.");
4 } else {
5     console.log("Você é menor de idade.");
6 }
```

Exemplo de Código 2.4: Estrutura if-else

Neste exemplo, a condição `idade >= 18` é avaliada. Se for verdadeira, o bloco de código dentro do if é executado e a mensagem **Você é maior de idade** é exibida no console. Caso contrário, o bloco de código dentro do else é executado e a mensagem **Você é menor de idade** é exibida.

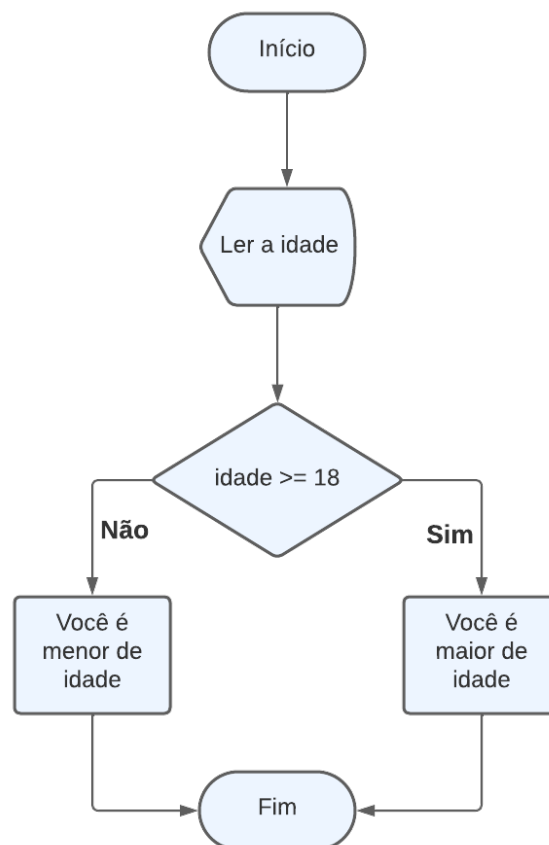


Figura 2.1: Diagrama de blocos sobre a estrutura if-else.

A estrutura do comando if-else permite que você escolha entre duas opções exclusivas com base em uma condição. Se a condição for verdadeira, o bloco de código dentro do if é executado; caso contrário, o bloco de código dentro do else é executado.

Para um melhor entendimento da estrutura você pode considerar ainda a figura 2.1 a qual apresenta o exemplo de código 2.4 utilizando diagrama de blocos ¹.

2.3 Estrutura if-elseif-else

Este tipo de estrutura permite que sejam realizadas múltiplas verificações antes de executar um bloco de código. Isso pode ser útil em casos em que há diferentes condições a serem verificadas e diferentes ações a serem tomadas dependendo das condições.

A estrutura básica da estrutura if-elseif-else é a seguinte:

¹ modelo padronizado e eficaz de representar as diversas direções que um algoritmo em seus passos lógicos pode tomar.

```
1 if (condição1) {
2     // bloco de código executado se condição1 for verdadeira
3 } else if (condição2) {
4     // bloco de código executado se condição1 for falsa e condição2 for
        verdadeira
5 } else if (condição3) {
6     // bloco de código executado se condição1 e condição2 forem falsas e
        condição3 for verdadeira
7 } else {
8     // bloco de código executado se nenhuma das condicoes anteriores for
        verdadeira
9 }
```

Exemplo de Código 2.5: Sintaxe da estrutura if-elseif-else

Observe que o bloco de código é executado apenas se a condição correspondente for verdadeira. Caso contrário, o programa passa para a próxima condição (se houver) ou executa o bloco de código no final, caso nenhuma condição seja verdadeira.

```
1 const idade = parseInt(prompt("Digite a idade: "));
2 if (idade >= 18) {
3     console.log("Você é adulto");
4 } else if (idade >= 12) {
5     console.log("Você é adolescente");
6 } else {
7     console.log("Você é criança");
8 }
```

Exemplo de Código 2.6: Estrutura if-elseif-else

Neste exemplo, a condição `idade >= 18` é avaliada. Se for verdadeira, o bloco de código dentro do **if** é executado e a mensagem **Você é adulto** é exibida no console. Caso contrário, o bloco de código dentro do **else if**, com a condição `idade >= 12` é avaliada. Se for verdadeira, o bloco de código dentro do **else if** é executado e a mensagem **Você é adolescente** é exibida no console. Em último caso, o bloco de código dentro do **else** é executado e a mensagem **Você é criança** é exibida no console.

A figura 2.2 mostra o diagrama de blocos para o exemplo de código 2.6:

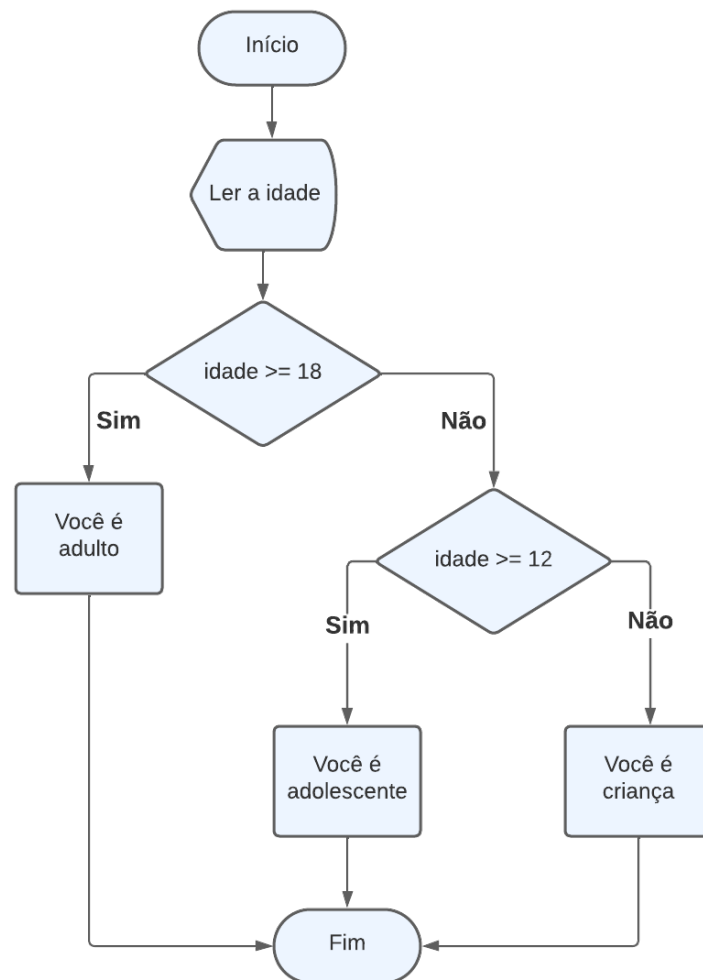


Figura 2.2: Diagrama de blocos sobre a estrutura if-elseif-else

2.4 Comandos de Decisão Aninhados

Os comandos de decisão aninhados podem ser utilizados para realizar verificações mais complexas, envolvendo múltiplas condições, em situações que uma subcondição só faz sentido caso outra condição mais geral seja satisfeita. Por exemplo:

```
1 const idade = parseInt(prompt("Digite a idade: "));
2 if (idade >= 18) {
3     const temCNH = prompt("Possui CNH (Sim/Não): ");
4     if (temCNH == "Sim") {
5         console.log("Pode dirigir");
```

```
6   } else {  
7       console.log("Não pode dirigir sem CNH");  
8   }  
9 } else {  
10    console.log("Não pode dirigir");  
11 }
```

Exemplo de Código 2.7: Comandos de decisão aninhados.

Nesse exemplo, é verificado se a idade é maior ou igual a 18 anos. Somente se sim, é solicitada a informação sobre a CNH e realizada uma nova verificação para saber se a pessoa tem carteira de habilitação (CNH). Se a pessoa tiver CNH, a mensagem **Pode dirigir** é exibida. Caso contrário, a mensagem **Não pode dirigir sem CNH** é exibida. Se a idade for menor que 18 anos, a mensagem **Não pode dirigir** é exibida.

Os comandos de decisão aninhados podem ser encadeados, criando uma estrutura mais complexa. No entanto, é importante tomar cuidado para não criar uma estrutura muito complexa e difícil de entender. Em alguns casos, é possível simplificar a estrutura utilizando operadores lógicos.

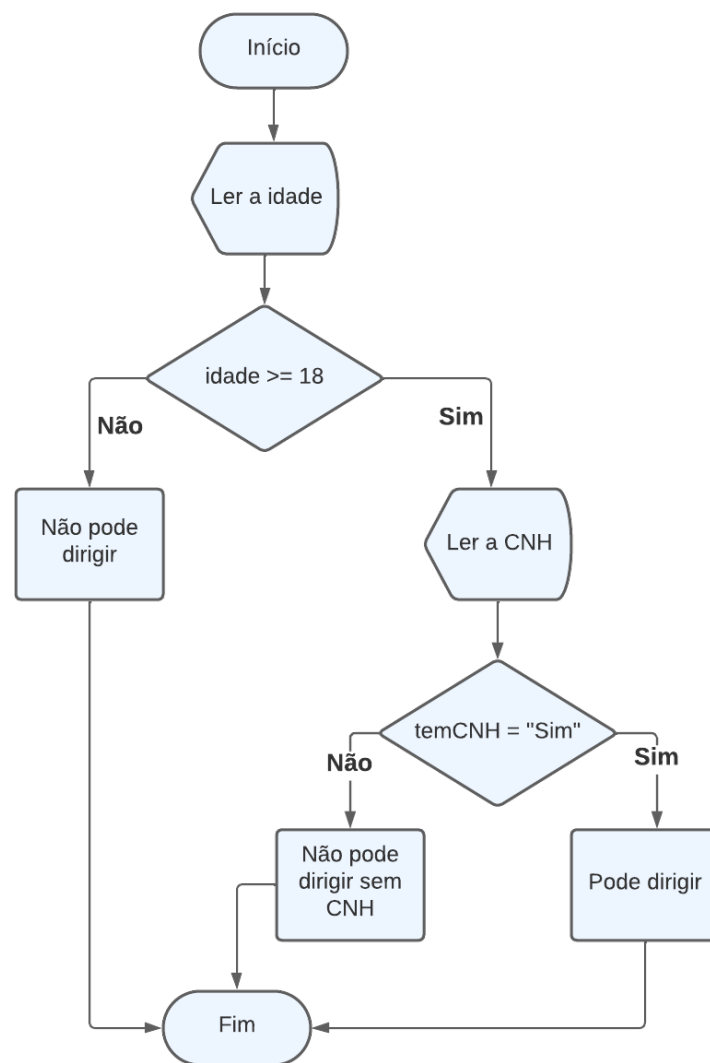


Figura 2.3: Diagrama de blocos sobre comandos de decisão aninhados

2.5 Exercícios Propostos

Exercício 2.1 — Comandos de Decisão: Exercício 01. Crie um algoritmo que receba pelo teclado o nome de um aluno e três notas. Ao final, deverá ser exibido o nome do aluno, sua média e o resultado (se for acima ou igual a 6, o aluno estará “aprovado”; se não for, estará “reprovado”). ■

Exercício 2.2 — Comandos de Decisão: Exercício 02. Usando o algoritmo do item 2, altere o resultado para: Média ≤ 3 , “reprovado”, Média < 6 , recuperação” e Média ≥ 6 , “aprovado”. ■

Exercício 2.3 — Comandos de Decisão: Exercício 03. Crie um algoritmo em que, dada a tabela a seguir, calcula o valor de desconto a ser concedido para um determinado cliente, de acordo com o valor da compra. O algoritmo deverá receber pelo teclado o nome do cliente e o valor total da compra:

Valor da compra	Porcentagem de desconto
Abaixo de R\$ 1.000,00	5
Entre R\$ 1.000,00 a R\$ 5.000,00	10
Acima de R\$ 5.000,00	15

Exercício 2.4 — Comandos de Decisão Aninhados: Exercício 01. Crie um algoritmo, utilizando a linguagem JavaScript que leia dois números. Caso os dois números sejam positivos você deve testá-los para exibir as seguintes frases:

- Os dois números são pares;
- Os dois números são ímpares;
- Um par e um ímpar.

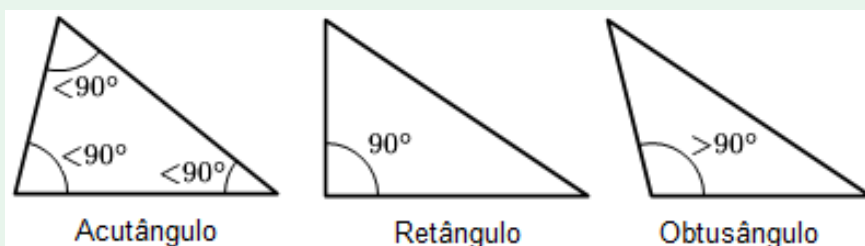
Se existir pelo menos um número negativo ou igual a zero, exiba a frase informando: Existe pelo menos um número 0 ou negativo! ■

Exercício 2.5 — Comandos de Decisão Aninhados: Exercício 02. Crie um algoritmo, utilizando a linguagem JavaScript, que receba pelo teclado os valores dos três ângulos internos de um triângulo. Depois verifique se é triângulo, de acordo com a Propriedade. Caso não seja triângulo mostre uma mensagem “Não é triângulo!”. Caso seja triângulo, mostre a classificação do triângulo quanto aos ângulos:

- **Acutângulo** (três ângulos agudos);
- **Retângulo** (um ângulo reto);
- **Obtusângulo** (um ângulo obtuso).

Propriedades:

- A soma dos ângulos de um triângulo deve ser igual a 180 graus;
- Ângulo agudo: menor do que 90 graus;
- Ângulo reto: exatamente 90 graus;
- Ângulo obtuso: maior que 90 graus e menor que 180 graus.



2.6 Gabarito dos Exercícios

Gabarito 2.1 — Comandos de Decisão: Exercício 01.



■

Gabarito 2.2 — Comandos de Decisão: Exercício 02.



■

Gabarito 2.3 — Comandos de Decisão: Exercício 03.



■

Gabarito 2.4 — Comandos de Decisão Aninhados: Exercício 01.



■

Gabarito 2.5 — Comandos de Decisão Aninhados: Exercício 02.



■

3. Comandos de Repetição

Os comandos de repetição são estruturas fundamentais na programação que permitem executar um bloco de código várias vezes. Eles desempenham um papel crucial quando precisamos automatizar tarefas repetitivas ou realizar operações em conjuntos de dados.

Existem diferentes tipos de comandos de repetição em JavaScript, como o laço `for`, o laço `while` e o laço `do while`. Cada um desses laços tem suas particularidades e é adequado para diferentes situações. O laço `for` é amplamente utilizado quando sabemos exatamente quantas vezes queremos repetir um bloco de código. Ele consiste em uma inicialização, uma condição e uma atualização, permitindo um controle preciso sobre o número de iterações.

O laço `while` é útil quando queremos repetir um bloco de código enquanto uma determinada condição `for` verdadeira. A condição é verificada antes de cada iteração, e se `for` falsa desde o início, o bloco de código não será executado.

Já o laço `do while` é semelhante ao laço `while`, porém a condição é verificada após a primeira execução do bloco de código. Isso garante que o bloco seja executado pelo menos uma vez, mesmo que a condição seja falsa inicialmente.

Além dos laços de repetição, também temos as instruções de controle de laço, como o `break` e o `continue`. O `break` é usado para interromper imediatamente a execução do laço e sair dele, enquanto o `continue` pula a iteração atual e continua para a próxima.

Ao utilizar comandos de repetição de forma adequada, podemos otimizar nosso código, tornando-o mais eficiente, conciso e fácil de entender. Eles nos permitem manipular listas, percorrer arrays, executar cálculos em série e muito mais. A compreensão e o domínio dos comandos de repetição são essenciais para se tornar um programador eficiente e produtivo.

3.1 Laço for

O laço for é uma estrutura de repetição muito utilizada em JavaScript. Ele permite executar um bloco de código repetidamente por um número específico de vezes. A estrutura básica do laço for é a seguinte:

```
1 for (inicialização; condição; atualização) {  
2     // bloco de código a ser repetido  
3 }
```

Exemplo de Código 3.1: Sintaxe do laço for

Na inicialização, declaramos e/ou atribuímos um valor à variável de controle do loop. A condição é uma expressão que determina se o loop deve continuar a ser executado. Enquanto a condição for verdadeira, o bloco de código dentro do laço é executado. Após a execução do bloco, a atualização é realizada, normalmente incrementando ou decrementando a variável de controle. Em seguida, a condição é avaliada novamente e o processo se repete até que a condição seja falsa.

Exemplo de uso do laço for:

```
1 for (let i = 0; i < 5; i++) {  
2     console.log(i);  
3 }
```

Exemplo de Código 3.2: Laço for

Nesse exemplo, o bloco de código dentro do laço for será executado 5 vezes. A cada iteração, o valor de *i* é exibido no console. A saída gerada será:

```
0  
1  
2  
3  
4
```

3.2 Laço while

O laço while é outra estrutura de repetição em JavaScript. Diferentemente do laço for, o laço while executa um bloco de código enquanto uma condição específica for verdadeira. A estrutura básica do laço while é a seguinte:

```
1 while (condição) {  
2     // bloco de código a ser repetido  
3 }
```

Exemplo de Código 3.3: Sintaxe do laço while

Enquanto a condição for verdadeira, o bloco de código é executado. A condição é avaliada antes de cada execução do bloco. Se a condição for falsa desde o início, o bloco não será executado.

Exemplo de uso do laço while:

```
1 let i = 0;  
2 while (i < 5) {  
3     console.log(i);  
4     i++;  
5 }
```

Exemplo de Código 3.4: Laço while

Nesse exemplo, o bloco de código dentro do laço while será executado enquanto **i** for menor que 5. A cada iteração, o valor de **i** é exibido no console e o valor de **i** é incrementado. A saída gerada será a mesma do exemplo anterior:

```
0  
1  
2  
3  
4
```

3.3 Laço do ... while

O laço do while é semelhante ao laço while, mas a condição é verificada no final do bloco de código. Isso significa que o bloco de código é executado pelo menos uma vez, mesmo que a condição seja falsa inicialmente. A estrutura básica do laço do while é a seguinte:

```
1 do {  
2     // bloco de código a ser repetido  
3 } while (condição);
```

Exemplo de Código 3.5: Sintaxe do laço do ... while

Exemplo de uso do laço do while:

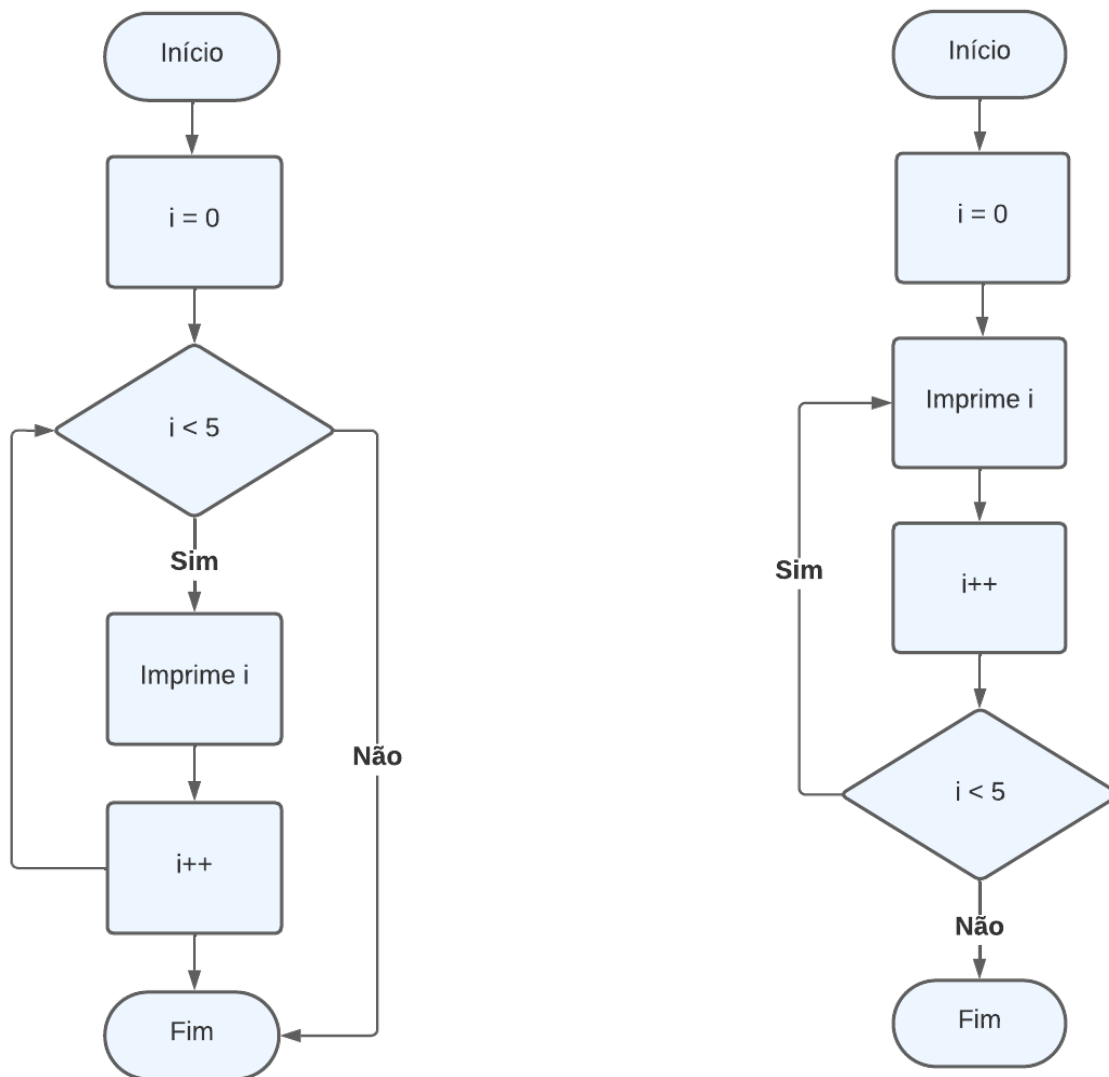
```
1 let i = 0;  
2 do {  
3     console.log(i);  
4     i++;  
5 } while (i < 5);
```

Exemplo de Código 3.6: Laço do ... while

Nesse exemplo, o bloco de código dentro do laço do while será executado pelo menos uma vez, já que a condição é verificada após a primeira execução do bloco. Em seguida, a condição é verificada novamente e o processo se repete enquanto a condição for verdadeira. A saída gerada será a mesma dos exemplos anteriores:

```
0  
1  
2  
3  
4
```

A figura 3.1, a seguir, compara visualmente o comportamento das estruturas de repetição nos códigos apresentados anteriormente (3.2, 3.4, 3.6):



(a) Diagrama de blocos dos comandos **for** e **while**

(b) Diagrama de blocos dos comandos **do...while**

Figura 3.1: Comparativo entre os comandos de repetição

3.4 Controle de laço: break e continue

Dentro de um laço de repetição, podemos utilizar as instruções **break** e **continue** para controlar o fluxo de execução.

A instrução **break** é utilizada para interromper imediatamente a execução do laço de repetição, independentemente da condição. Ao encontrar a instrução **break**, o programa sai do laço e continua a execução a partir do próximo comando após o laço.

```
1 for (let i = 0; i < 5; i++) {  
2     if (i === 3) {  
3         break;  
4     }  
5     console.log(i);  
6 }
```

Exemplo de Código 3.7: Controle de laço com **break**

Nesse exemplo, o laço for será interrompido quando **i** tiver o valor 3. Portanto, apenas os valores 0, 1 e 2 serão exibidos no console:

```
0  
1  
2
```

A instrução **continue** é utilizada para pular a iteração atual do laço de repetição e continuar para a próxima iteração, ignorando o restante do bloco de código.

```
1 for (let i = 0; i < 5; i++) {  
2     if (i === 2) {  
3         continue;  
4     }  
5     console.log(i);  
6 }
```

Exemplo de Código 3.8: Controle de laço com **continue**

Nesse exemplo, quando **i** tiver o valor 2, a instrução **continue** será encontrada e a iteração atual será pulada. Portanto, o valor 2 não será exibido no console, mas os outros valores serão exibidos normalmente.

```
0  
1  
3  
4
```

As instruções **break** e **continue** podem ser úteis em determinadas situações para controlar o fluxo de execução e tomar decisões dentro de um laço de repetição.

3.5 Comandos de Repetição Aninhados

Os comandos de repetição aninhados são úteis quando precisamos realizar operações repetidas em estruturas de dados multidimensionais, como matrizes (arrays) ou tabelas. Essas estruturas de dados possuem múltiplas dimensões, como linhas e colunas, e os comandos de repetição aninhados nos permitem percorrer todas as combinações possíveis dessas dimensões.

A estrutura básica dos comandos de repetição aninhados é a seguinte:

```
1 for (inicialização; condição; atualização) {  
2     // bloco de código a ser repetido  
3     for (inicialização; condição; atualização) {  
4         // bloco de código a ser repetido  
5     }  
6 }
```

Exemplo de Código 3.9: Sintaxe dos comandos de repetição aninhados

Dentro do primeiro comando de repetição, podemos ter outro comando de repetição e assim por diante.

Cada comando de repetição aninhado terá sua própria inicialização, condição e atualização.

A fim de ilustrar o uso dos comandos de repetição aninhados, considere o seguinte exemplo:

```
1 for (let i = 1; i <= 5; i++) {  
2     for (let j = 1; j <= 5; j++) {  
3         console.log(i + " x " + j + " = " + (i * j));  
4     }  
5     console.log(); // Imprime quebra de linha  
6 }
```

Exemplo de Código 3.10: Comandos de repetição aninhados

Nesse exemplo, temos dois laços for aninhados. O primeiro laço for é responsável por percorrer os valores de **i** de 1 a 5, enquanto o segundo laço for percorre os valores de **j** de 1 a 5. Para cada combinação de valores **i** e **j**, é exibida a mensagem indicando a multiplicação entre eles.

A saída gerada por esse código será a seguinte:

$$1 \times 1 = 1$$

$$1 \times 2 = 2$$

$$1 \times 3 = 3$$

$$1 \times 4 = 4$$

$$1 \times 5 = 5$$

$$2 \times 1 = 2$$

$$2 \times 2 = 4$$

$$2 \times 3 = 6$$

$$2 \times 4 = 8$$

$$2 \times 5 = 10$$

$$3 \times 1 = 3$$

$$3 \times 2 = 6$$

$$3 \times 3 = 9$$

$$3 \times 4 = 12$$

$$3 \times 5 = 15$$

$$4 \times 1 = 4$$

$$4 \times 2 = 8$$

$$4 \times 3 = 12$$

$$4 \times 4 = 16$$

$$4 \times 5 = 20$$

$$5 \times 1 = 5$$

$$5 \times 2 = 10$$

$$5 \times 3 = 15$$

$$5 \times 4 = 20$$

$$5 \times 5 = 25$$

Esse exemplo demonstra como os comandos de repetição aninhados podem ser utilizados para percorrer e executar operações em diferentes combinações de elementos em estruturas de dados multidimensionais.

A figura 3.2, a seguir, mostra o diagrama de blocos para o exemplo de código 3.10, apresentado anteriormente:

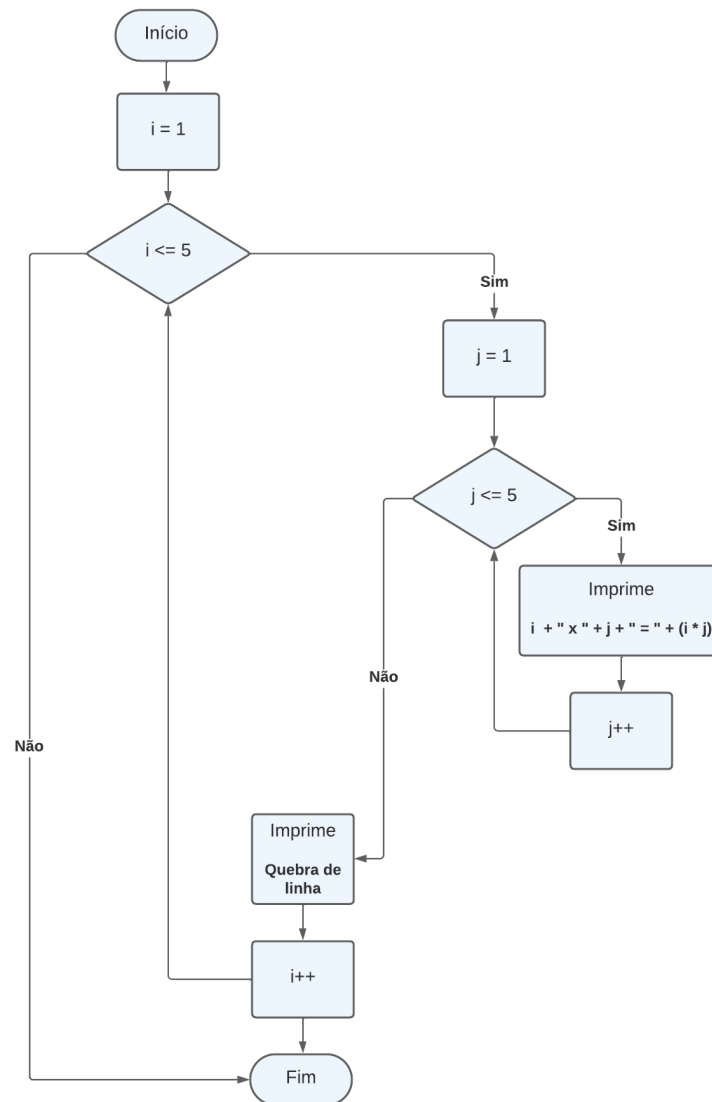


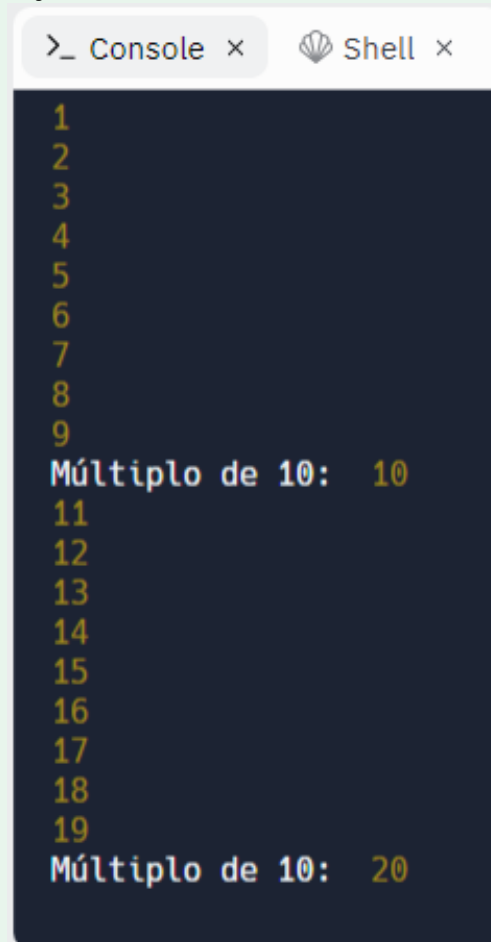
Figura 3.2: Diagrama de blocos sobre comandos de repetição aninhados

Lembre-se de que os comandos de repetição aninhados podem aumentar a complexidade e a legibilidade do código. Portanto, é importante utilizá-los de forma adequada e considerar alternativas, como o uso de funções e abordagens mais modulares, quando apropriado.

3.6 Exercícios Propostos

Exercício 3.1 — Comandos de Repetição: Exercício 01. Desenvolva um programa imprime na tela os números entre 7 e 1000 que tem resto 3 quando divididos por 7. ■

Exercício 3.2 — Comandos de Repetição: Exercício 02. Faça um algoritmo que conte de 1 a 100 e a cada múltiplo de 10 emita uma mensagem: “Múltiplo de 10: <número>”. Na figura a seguir temos uma parte da exibição do resultado (neste caso, considerando o intervalo de 1 até 20):



```
>_ Console x Shell x
1
2
3
4
5
6
7
8
9
Múltiplo de 10: 10
11
12
13
14
15
16
17
18
19
Múltiplo de 10: 20
```

Exercício 3.3 — Comandos de Repetição: Exercício 03. Desenvolva um algoritmo que obtém números do usuário e os soma. A cada repetição algoritmo deve perguntar ao usuário se o mesmo deseja continuar a digitar números. Enquanto o usuário digitar "s" o algoritmo continua a receber números e somá-los. Quando o usuário digita qualquer outra coisa o algoritmo termina e apresenta o valor da soma dos números. ■

Exercício 3.4 — Comandos de Repetição: Exercício 04. Faça um algoritmo que determine o maior entre N números inteiros positivos. A condição de parada é a entrada de um valor 0, ou seja, o algoritmo deve ficar calculando o maior até que a entrada seja igual a 0 (ZERO). ■

Exercício 3.5 — Comandos de Repetição Aninhados: Exercício 01. Escreva um algoritmo que lê via teclado um número inteiro positivo e mostre na tela, como resultado, a quantidade de números primos existentes entre 1 e n. ■

Exercício 3.6 — Comandos de Repetição Aninhados: Exercício 02. Escreva um programa que mostre na tela os 3 primeiros números perfeitos. Um número perfeito é aquele que é igual à soma dos seus divisores. Exemplos de números perfeitos: $6 = 1+2+3$; $28 = 1+2+4+7+14$; etc. ■

3.7 Gabarito dos Exercícios

Gabarito 3.1 — Comandos de Repetição: Exercício 01.



■

Gabarito 3.2 — Comandos de Repetição: Exercício 02.



■

Gabarito 3.3 — Comandos de Repetição: Exercício 03.



■

Gabarito 3.4 — Comandos de Repetição: Exercício 04.



■

Gabarito 3.5 — Comandos de Repetição Aninhados: Exercício 01.



■

Gabarito 3.6 — Comandos de Repetição Aninhados: Exercício 02.



■

4. Coleções de dados (Arrays)

Arrays são estruturas de dados em JavaScript que nos permitem armazenar e acessar múltiplos valores em uma única variável. Eles são coleções indexadas, o que significa que os elementos do array são numerados sequencialmente a partir de zero.

4.1 Criando um Array

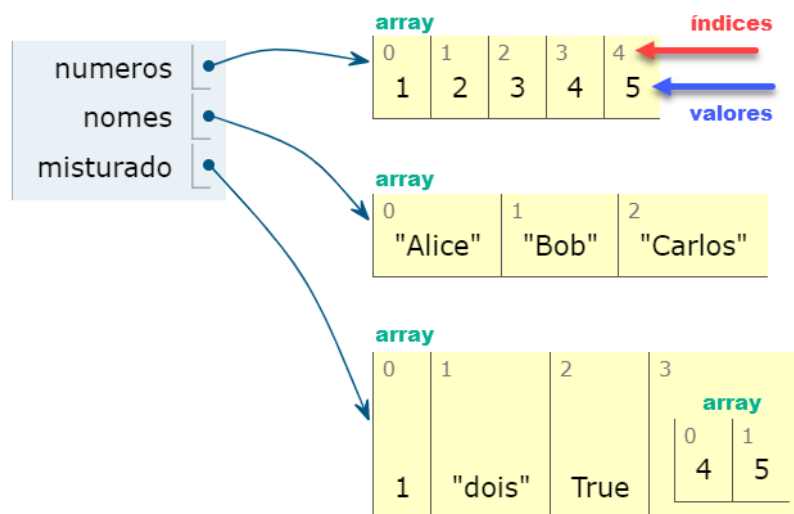
Podemos criar um array em JavaScript de diferentes maneiras. Aqui estão alguns exemplos:

```
1 // Array vazio
2 const arrayVazio = [];
3
4 // Array com elementos
5 const numeros = [1, 2, 3, 4, 5];
6 const nomes = ["Alice", "Bob", "Carlos"];
7 const misturado = [1, "dois", true, [4, 5]];
```

Exemplo de Código 4.1: Criando um array

Podemos criar um array vazio simplesmente utilizando `[]`, ou podemos especificar os elementos do array separados por vírgula dentro dos colchetes.

A figura 4.1 apresenta graficamente os arrays **numeros**, **nomes** e **misturado**:

**Figura 4.1:** Visualização de Arrays.

4.2 Referenciando um Array

Para acessar os elementos de um array, utilizamos o nome do array seguido pelo índice do elemento entre colchetes. O índice do primeiro elemento é sempre 0.

```
1 const numeros = [10, 20, 30, 40, 50];
2 console.log(numeros[0]); // Resultado: 10
3 console.log(numeros[2]); // Resultado: 30
```

Exemplo de Código 4.2: Referenciando um array

No exemplo acima, o valor **10** é acessado utilizando `numeros[0]` e o valor **30** é acessado utilizando `numeros[2]`.

4.3 Iterando um Array

Podemos percorrer os elementos de um array utilizando loops, como o **for** ou o **for...of**. Aqui está um exemplo com o **for**:

```
1 const numeros = [1, 2, 3, 4, 5];
2 for (let i = 0; i < numeros.length; i++) {
3   console.log(numeros[i]);
4 }
```

Exemplo de Código 4.3: Iterando um array com for

Nesse exemplo, utilizamos o **for** para percorrer cada elemento do array **numeros** e exibi-los no console.

Também podemos utilizar o **for...of** para percorrer os elementos de um array:

```
1 const numeros = [1, 2, 3, 4, 5];
2 for (let numero of numeros) {
3   console.log(numero);
4 }
```

Exemplo de Código 4.4: Iterando um array com **for...of**

O **for...of** percorre automaticamente cada elemento do array, atribuindo-o à variável **numero**, que utilizamos para exibir o valor no console.

Nos dois exemplos de trechos de códigos anteriores a saída é a mesma:

```
1
2
3
4
5
```

4.4 Métodos de um Array

Os arrays em JavaScript possuem diversos métodos embutidos que facilitam a manipulação e transformação dos elementos. Apresentamos uma perspectiva visual de alguns destes métodos:

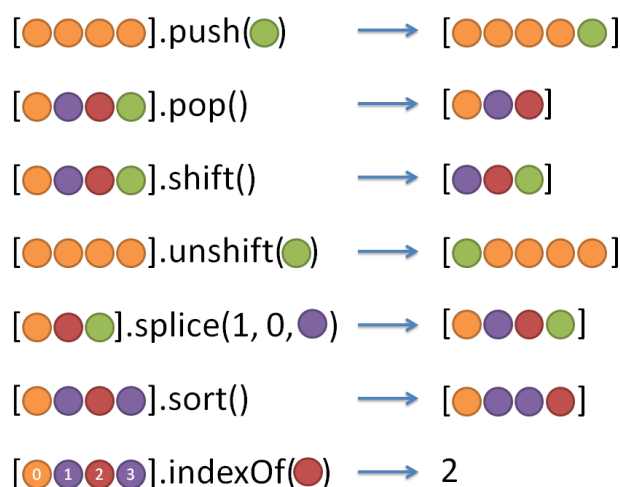


Figura 4.2: Visualização dos métodos de Arrays.

push(): Adiciona um ou mais elementos ao final do array.

```
1 const numeros = [1, 2, 3];
2 numeros.push(4);
3 console.log(numeros); // Resultado: [1, 2, 3, 4]
```

Exemplo de Código 4.5: Método **push** de um array

pop(): Remove o último elemento do array e retorna esse elemento.

```
1 const numeros = [1, 2, 3];
2 const ultimoElemento = numeros.pop();
3 console.log(numeros); // Resultado: [1, 2]
4 console.log(ultimoElemento); // Resultado: 3
```

Exemplo de Código 4.6: Método **pop** de um array

shift(): Remove o primeiro elemento do array e retorna esse elemento.

```
1 const numeros = [1, 2, 3];
2 const primeiroElemento = numeros.shift();
3 console.log(numeros); // Resultado: [2, 3]
4 console.log(primeiroElemento); // Resultado: 1
```

Exemplo de Código 4.7: Método **shift** de um array

unshift(): Adiciona um ou mais elementos ao início do array.

```
1 const numeros = [2, 3];
2 numeros.unshift(1);
3 console.log(numeros); // Resultado: [1, 2, 3]
```

Exemplo de Código 4.8: Método **unshift** de um array

splice(): Permite adicionar, remover ou substituir elementos em qualquer posição do array.

```
1 const numeros = [1, 2, 3, 4, 5];
2 numeros.splice(2, 1, 10); // Remove 1 elemento a partir do índice 2 e
   insere o valor 10 no mesmo índice.
3 console.log(numeros); // Resultado: [1, 2, 10, 4, 5]
```

Exemplo de Código 4.9: Método **splice** de um array

sort(): Ordena os elementos do array em ordem crescente.

```
1 const numeros = [3, 1, 4, 2, 5];  
2 numeros.sort();  
3 console.log(numeros); // Resultado: [1, 2, 3, 4, 5]
```

Exemplo de Código 4.10: Método **sort** de um array

indexOf(): Retorna o primeiro índice em que um elemento específico é encontrado no array.

```
1 const numeros = [10, 20, 30, 40, 50];  
2 const indice = numeros.indexOf(30);  
3 console.log(indice); // Resultado: 2
```

Exemplo de Código 4.11: Método **indexOf** de um array

Esses são apenas alguns exemplos dos métodos disponíveis para manipular arrays em JavaScript. Existem muitos outros métodos úteis que você pode explorar para trabalhar com coleções de dados de forma eficiente e flexível.

4.5 Exercícios Propostos

Exercício 4.1 — Coleções de dados (Arrays): Exercício 01. Leia 8 números inteiros do usuário e armazene-os em uma coleção indexada (array). Depois, calcule a média desses valores. Ao final, exiba todos os números do array maiores que a média calculada.

Entrada	Saída
array = [9, 4, 3, 7, 10, 6, 8, 5]	9 7 10 8

■

Exercício 4.2 — Coleções de dados (Arrays): Exercício 02. Dadas duas sequências com 4 números inteiros entre 0 e 9, calcular a sequência de números que representa a soma das sequências anteriores.

Entrada	Saída
array1 = [1, 2, 3, 4]	[6, 8, 10, 12]
array2 = [5, 6, 7, 8]	

■

Exercício 4.3 — Coleções de dados (Arrays): Exercício 03. Escreva um algoritmo que calcule a interseção (valores em comum) entre os valores contidos em dois arrays de 5 elementos, array1 e array2 (lidos pelo teclado) e armazene estes valores no array3, mostrando-os no final.

Entrada	Saída
array1 = [1, 14, 2, 4, 7]	[1, 2]
array2 = [5, 6, 1, 2, 10]	

■

4.6 Gabarito dos Exercícios

Gabarito 4.1 — Coleções de dados (Arrays): Exercício 01.



■

Gabarito 4.2 — Coleções de dados (Arrays): Exercício 02.



■

Gabarito 4.3 — Coleções de dados (Arrays): Exercício 03.



■

5. Funções e Procedimentos

Funções e Procedimentos são elementos fundamentais na programação, pois nos permitem organizar o código em blocos reutilizáveis, modularizando as tarefas e facilitando o desenvolvimento e a manutenção do código. Embora compartilhem semelhanças, funções e procedimentos possuem algumas diferenças importantes.

Uma função é um bloco de código que recebe uma entrada, realiza um processamento e retorna um resultado. Ela pode receber parâmetros como entrada e pode retornar um valor específico após o processamento. As funções são especialmente úteis quando queremos realizar uma operação específica e obter um resultado desejado. Elas podem ser definidas utilizando a palavra-chave **function** seguida do nome da função, parâmetros entre parênteses e um bloco de código entre chaves.

Por outro lado, um procedimento também é um bloco de código, mas não retorna um valor específico. Ele é utilizado para executar uma sequência de ações sem a necessidade de retornar um resultado. Em vez de retornar um valor, um procedimento pode, por exemplo, realizar ações como exibir uma mensagem na tela ou alterar o valor de uma variável.

Para utilizar uma função, podemos chamá-la em nosso código, passando os argumentos necessários. A chamada de função é feita pelo nome da função seguido por parênteses contendo os argumentos que serão passados à função. Os argumentos são os valores que serão utilizados pela função durante sua execução.

Em alguns casos, podemos definir argumentos padrão para uma função, o que significa que eles terão um valor predefinido caso não sejam passados como argumentos na chamada da função. Isso nos proporciona flexibilidade ao utilizar a função, permitindo que ela seja chamada com menos parâmetros, assumindo valores padrão para os parâmetros não fornecidos.

Esses são apenas alguns conceitos introdutórios sobre funções e procedimentos. No restante dos tópicos, exploraremos exemplos de código para cada um deles, mostrando como definir e utilizar funções, além de abordar o escopo de variáveis, expressões de função e as compactas Arrow Functions. Continue lendo para aprofundar seu conhecimento sobre essa importante parte da programação em JavaScript.

5.1 Definição de Função

A definição de uma função em JavaScript envolve a criação de um bloco de código que pode ser reutilizado em diferentes partes do programa. Uma função é definida utilizando a palavra-chave ‘function’, seguida pelo nome da função, parênteses contendo os parâmetros da função (opcionais) e um bloco de código entre chaves.

Exemplo de definição de função:

```
1 function saudacao(nome) {  
2     console.log("Olá, " + nome + "!");  
3 }
```

Exemplo de Código 5.1: Definição de função

Nesse exemplo, definimos uma função chamada **saudacao** que recebe um parâmetro **nome**. Dentro da função, utilizamos o ‘console.log’ para exibir uma mensagem de saudação no console.

5.2 Chamada de Função

Uma vez que uma função está definida, podemos chamá-la em diferentes partes do programa. A chamada de função é feita pelo nome da função seguido por parênteses, contendo os argumentos que serão passados para a função.

Exemplo de chamada de função:

```
1 saudacao("Alice"); // Saída: "Olá, Alice!"
```

Exemplo de Código 5.2: Chamada de função

Nesse exemplo, chamamos a função **saudacao** passando o valor **"Alice"** como argumento. A função é executada e a mensagem **"Olá, Alice!"** é exibida no console.

5.3 Argumentos Default

Em JavaScript, podemos definir valores padrão para os parâmetros de uma função. Isso significa que se um argumento não for fornecido durante a chamada da função, o valor padrão será utilizado em seu lugar.

Exemplo de argumentos default:

```
1 function saudacao(nome = "Visitante") {  
2     console.log("Olá, " + nome + "!");  
3 }  
4  
5 saudacao(); // Saída: "Olá, Visitante!"  
6 saudacao("Bob"); // Saída: "Olá, Bob!"
```

Exemplo de Código 5.3: Argumentos Default

Nesse exemplo, definimos um valor padrão **"Visitante"** para o parâmetro **nome**. Se nenhum argumento for passado, o valor padrão será utilizado. Caso contrário, o valor passado será utilizado.

5.4 Escopo de Variáveis

O escopo de variáveis em JavaScript determina onde uma variável pode ser acessada. Variáveis declaradas dentro de uma função têm **escopo local**, ou seja, são acessíveis apenas dentro da função. Variáveis declaradas fora de qualquer função têm **escopo global**, sendo acessíveis em todo o programa.

Exemplo de escopo de variáveis:

```
1 function somar(a, b) {  
2     const resultado = a + b;  
3     console.log(resultado); // Saída: resultado da soma  
4 }  
5  
6 somar(3, 4); // Resultado: 7  
7 console.log(resultado); // Erro: resultado is not defined
```

Exemplo de Código 5.4: Escopo de variáveis de função

Nesse exemplo, a variável **resultado** é declarada dentro da função **somar** e é acessível apenas dentro dessa função. Se tentarmos acessar a variável fora da função, ocorrerá um erro.



Para lembrar das diferenças quanto ao escopo das diferentes possibilidades de declarações de variáveis (**var**, **let**, **const**) leia novamente a subseção 1.3

5.5 Expressão de Função

Além de definir funções utilizando a declaração **function**, em JavaScript também é possível utilizar expressões de função. Uma expressão de função é uma função atribuída a uma variável ou a uma constante.

Exemplo de expressão de função:

```
1 const saudacao = function(nome) {  
2   console.log("Olá, " + nome + "!");  
3 };  
4  
5 saudacao("Carol"); // Saída: "Olá, Carol!"
```

Exemplo de Código 5.5: Expressão de função

Nesse exemplo, utilizamos uma expressão de função para criar uma função anônima e atribuí-la à variável **saudacao**. Em seguida, chamamos a função passando o argumento **Carol**.

5.6 Arrow Functions

As Arrow Functions são uma forma mais compacta de definir funções em JavaScript, introduzida na versão ES6 da linguagem. Elas são definidas utilizando a sintaxe de seta **=>** e têm algumas características especiais, como a ausência de escopo próprio.

Exemplo de Arrow Function:

```
1 const saudacao = (nome) => {  
2   console.log("Olá, " + nome + "!");  
3 };  
4  
5 saudacao("Daniel"); // Saída: "Olá, Daniel!"
```

Exemplo de Código 5.6: Arrow functions

Nesse exemplo, utilizamos uma Arrow Function para definir a função **saudacao**. A sintaxe mais concisa torna a leitura e a escrita do código mais simples.

5.7 Comparativo entre os tipos de funções

O exemplo de código 5.7 apresenta uma comparação entre as três formas vistas anteriormente de se escrever funções (5.1 Definição de função, 5.5 Expressão de função e 5.6 Arrow function):

```
1 // declaração da função
2 function somar1(n1, n2) {
3     return n1 + n2;
4 }
5
6 // declaração da função como expressão
7 const somar2 = function(n1, n2) { return n1 + n2; };
8
9 // declaração da arrow function
10 const somar3 = (n1, n2) => n1 + n2;
11
12 // chamada das funções
13 console.log(somar1(1, 2));
14 console.log(somar2(1, 2));
15 console.log(somar3(1, 2));
```

Exemplo de Código 5.7: Comparativo entre os tipos de funções

Nas três chamadas de funções a saída é a mesma:

```
3
3
3
```

5.8 Exercícios Propostos

Exercício 5.1 — Funções e Procedimentos: Exercício 01. O Índice de Massa Corporal (IMC) é uma medida do grau de obesidade uma pessoa. Através do cálculo de IMC é possível saber se alguém está acima ou abaixo dos parâmetros ideais de peso para sua estatura. Calcular IMC requer a aplicação de uma fórmula que leva em conta seu peso e altura. IMC é igual ao peso dividido pelo quadrado da altura ($IMC = \text{peso} / (\text{altura} * \text{altura})$).

IMC	Código	Descrição
Abaixo de 18,5	1	Você está abaixo do peso ideal!
Entre 18,5 e 24,9	2	Parabéns! Você está em seu peso normal!
Entre 25,0 e 29,9	3	Você está acima de seu peso (sobrepeso)!
Acima 29,9	4	Obesidade!

a) Faça uma função para calcular o IMC. Esta função deve receber como parâmetros o peso e a altura de uma determinada pessoa, e retornar o IMC.

b) Faça outra função para verificar a situação. Esta função deve receber como parâmetro o IMC, e retornar um código (1, 2, 3 ou 4) de acordo com a situação da pessoa que possui estas medidas.

c) Faça um procedimento para exibir descrição da situação. Este procedimento deve receber como parâmetro o código da situação da pessoa, e exibir na tela o texto com a descrição da situação.

Considerando a existência dessas duas funções e do procedimento, o seu algoritmo deve receber dados de 5 pessoas com relação ao nome, peso e a altura. Imediatamente após a digitação dos dados de cada pessoa deve ser informada a descrição da sua situação com relação ao IMC. Exemplo:

```
>_ Console x Shell x +
PESSOA 1
Entre com o nome: João da Silva
Entre com o peso (Kg): 80
Entre com a altura (Metros): 1.75
Você está acima de seu peso (sobrepeso).

PESSOA 2
Entre com o nome: 
```

5.9 Gabarito dos Exercícios

Gabarito 5.1 — Funções e Procedimentos: Exercício 01.





Parte II: Práticas

6. Práticas Propostas

Nesta seção serão apresentadas diferentes versões de códigos relacionados a elaboração de algoritmos para solucionar o **problema de cálculo do salário líquido de um cidadão**. O código será aprimorado a cada nova versão, considerando evoluções nas perspectivas dos seguintes assuntos abordados na seção **Parte I: Conceitos**:

- Operadores (Seção 1.4);
- Entrada de Dados (Seção 1.5);
- Comandos de Decisão (Capítulo 2);
- Comandos de Decisão Aninhados (Seção 2.4);
- Comandos de Repetição (Capítulo 3);
- Comandos de Repetição Aninhados (Seção 3.5);
- Coleções do tipo Arrays (Capítulo 4);
- Funções e Procedimentos (Capítulo 5);

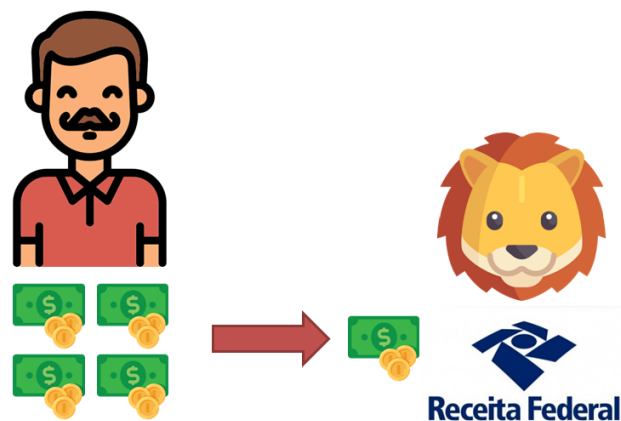


Figura 6.1: Problema do cálculo do salário líquido de um cidadão.

6.1 Operadores

6.1.1 Prática Proposta

Prática 6.1 — Operadores. Elabore um algoritmo para calcular o salário líquido de uma pessoa. O salário bruto dessa pessoa é de R\$ 7.500,00. O imposto de renda a ser descontado é equivalente a 10% do salário bruto. ■

6.1.2 Gabarito da Prática



```
1 var salário_bruto = 7500;
2 var ir = salario_bruto * 0.10;
3 var salario_liquido = salário_bruto - ir;
4 console.log("Salário líquido: R$", salario_liquido);
```

Exemplo de Código 6.1: Prática - Operadores

6.2 Entrada de Dados

6.2.1 Prática Proposta

Prática 6.2 — Entrada de Dados. Elabore um algoritmo para calcular o salário líquido de uma pessoa. Solicite ao usuário que digite seu nome e o valor de seu salário bruto. O imposto de renda a ser descontado é equivalente a 10% do salário bruto. Ao final, mostre na tela o valor do salário líquido. ■

6.2.2 Gabarito da Prática



```
1 const prompt = require("prompt-sync")();
2
3 var nome = prompt("Digite seu nome: ");
4 var salario_bruto = parseFloat(prompt("Digite seu salário: "));
5
6 var ir = salario_bruto * 0.10;
7 var salario_liquido = salario_bruto - ir;
8 console.log("Salário líquido: R$", salario_liquido);
```

Exemplo de Código 6.2: Prática - Entrada de Dados

6.3 Comandos de Decisão

6.3.1 Prática Proposta

Prática 6.3 — Comandos de Decisão. Elabore um algoritmo para calcular o salário líquido de uma pessoa. Solicite ao usuário que digite seu nome e o valor de seu salário bruto. O imposto de renda a ser descontado do salário bruto deve considerar as seguintes regras:

- salário bruto (de 0,00 até 1.903,98): 5%;
- salário bruto (de 1.903,99 até 2.826,65): 7,5%;
- salário bruto (a partir de 2.826,66): 15%.

Ao final, mostre na tela o valor do salário líquido.

6.3.2 Gabarito da Prática



```
1  const prompt = require("prompt-sync")();
2
3  var nome = prompt("Digite seu nome: ");
4  var salario_bruto = parseFloat(prompt("Digite seu salário bruto: "));
5
6  if (salario_bruto <= 1903.98){
7      var ir = salario_bruto * 0.05;
8  }else if(salario_bruto <= 2826.65){
9      var ir = salario_bruto * 0.075;
10 }else{
11     var ir = salario_bruto * 0.15;
12 }
13
14 var salario_liquido = salario_bruto - ir;
15 console.log("Salário líquido: R$", salario_liquido);
```

Exemplo de Código 6.3: Prática - Comandos de Decisão

6.4 Comandos de Decisão Aninhados

6.4.1 Prática Proposta

Prática 6.4 — Comandos de Decisão Aninhados. Elabore um algoritmo para calcular o salário líquido de uma pessoa. Solicite ao usuário que digite seu nome e o valor de seu salário bruto. Peça também, para o usuário digitar a quantidade de dependentes. Calcule a renda familiar per capita. Caso a renda para cada membro da família seja menor que R\$ 500,00 a pessoa ficará isenta de imposto de renda, ou seja, não será calculado dentre as faixas salariais e seu valor será zero. Caso a renda para cada membro da família seja maior ou igual a R\$ 500,00 o imposto de renda a ser descontado do salário bruto deve considerar as seguintes regras:

- salário bruto (de 0,00 até 1.903,98): 5%;
- salário bruto (de 1.903,99 até 2.826,65): 7,5%;
- salário bruto (a partir de 2.826,66): 15%.

Ao final, mostre na tela o valor do salário líquido.

6.4.2 Gabarito da Prática



```
1  const prompt = require("prompt-sync")();
2
3  var nome = prompt("Digite seu nome: ");
4  var salario_bruto = parseFloat(prompt("Digite seu salário bruto: "));
5  var dependentes = parseInt(prompt("Digite o número de dependentes: "));
6
7  var renda_percapta = salario_bruto / (dependentes + 1);
8
9  if (renda_percapta >= 500){
10
11     if (salario_bruto <= 1903.98){
12         var ir = salario_bruto * 0.05;
13     }else if(salario_bruto <= 2826.65){
```

```
14     var ir = salario_bruto * 0.075;
15 }else{
16     var ir = salario_bruto * 0.15;
17 }
18
19 }else{
20     var ir = 0;
21 }
22
23 var salario_liquido = salario_bruto - ir;
24 console.log("Salário líquido: R$", salario_liquido);
```

Exemplo de Código 6.4: Prática - Comandos de Decisão Aninhados

6.5 Comandos de Repetição

6.5.1 Prática Proposta

Prática 6.5 — Comandos de Repetição. Elabore um algoritmo para calcular o salário líquido de 5 pessoas. Solicite ao usuário que digite seu nome e o valor de seu salário bruto. Peça também, para o usuário digitar a quantidade de dependentes. Calcule a renda familiar per capita. Caso a renda para cada membro da família seja menor que R\$ 500,00 a pessoa ficará isenta de imposto de renda, ou seja, não será calculado dentre as faixas salariais e seu valor será zero. Caso a renda para cada membro da família seja maior ou igual a R\$ 500,00 o imposto de renda a ser descontado do salário bruto deve considerar as seguintes regras:

- salário bruto (de 0,00 até 1.903,98): 5%;
- salário bruto (de 1.903,99 até 2.826,65): 7,5%;
- salário bruto (a partir de 2.826,66): 15%.

Ao final, mostre na tela o valor do salário líquido.

6.5.2 Gabarito da Prática



```
1  const prompt = require("prompt-sync")();
2
3  for(var i = 1; i <= 5; i++){
4      console.log("Pessoa ", i);
5
6      var nome = prompt("Digite seu nome: ");
7      var salario_bruto = parseFloat(prompt("Digite seu salário bruto: "));
8      var dependentes = parseInt(prompt("Digite o número de dependentes: "));
9
10     var renda_percapta = salario_bruto / (dependentes + 1);
11
12     if (renda_percapta >= 500){
```

```
13
14     if (salario_bruto <= 1903.98){
15         var ir = salario_bruto * 0.05;
16     }else if(salario_bruto <= 2826.65){
17         var ir = salario_bruto * 0.075;
18     }else{
19         var ir = salario_bruto * 0.15;
20     }
21
22 }else{
23     var ir = 0;
24 }
25
26 var salario_liquido = salario_bruto - ir;
27 console.log("Salário líquido: R$", salario_liquido, "\n");
28 }
```

Exemplo de Código 6.5: Prática - Comandos de Repetição

6.6 Comandos de Repetição Aninhados

6.6.1 Prática Proposta

Prática 6.6 — Comandos de Repetição Aninhados. Elabore um algoritmo para calcular o salário líquido de 5 pessoas. Solicite ao usuário que digite seu nome e o valor de seu salário bruto. Peça também, para o usuário digitar a quantidade de dependentes. Para cada um dos dependentes deve ser solicitado o ganho mensal. Este valor deverá ser adicionado ao salário bruto. Calcule a renda familiar per capita. Caso a renda para cada membro da família seja menor que R\$ 500,00 a pessoa ficará isenta de imposto de renda, ou seja, não será calculado dentre as faixas salariais e seu valor será zero. Caso a renda para cada membro da família seja maior ou igual a R\$ 500,00 o imposto de renda a ser descontado do salário bruto deve considerar as seguintes regras:

- salário bruto (de 0,00 até 1.903,98): 5%;
- salário bruto (de 1.903,99 até 2.826,65): 7,5%;
- salário bruto (a partir de 2.826,66): 15%.

Ao final, mostre na tela o valor do salário líquido.

6.6.2 Gabarito da Prática



```
1  const prompt = require("prompt-sync")();
2
3  for(var i = 1; i <= 5; i++){
4      console.log("Pessoa ", i);
5
6      var nome = prompt("Digite seu nome: ");
7      var salario_bruto = parseFloat(prompt("Digite seu salário bruto: "));
8      var dependentes = parseInt(prompt("Digite o número de dependentes: "));
9
10     for(var j = 1; j <= dependentes; j++){
11         console.log("Dependente ", j);
```

```
12     var ganho = parseFloat(prompt("Digite seu ganho mensal: "));
13     salario_bruto = salario_bruto + ganho;
14 }
15
16 var renda_percapta = salario_bruto / (dependentes + 1);
17
18 if (renda_percapta >= 500){
19
20     if (salario_bruto <= 1903.98){
21         var ir = salario_bruto * 0.05;
22     }else if(salario_bruto <= 2826.65){
23         var ir = salario_bruto * 0.075;
24     }else{
25         var ir = salario_bruto * 0.15;
26     }
27
28 }else{
29     var ir = 0;
30 }
31
32 var salario_liquido = salario_bruto - ir;
33 console.log("Salário líquido: R$", salario_liquido, "\n");
34
35 }
```

Exemplo de Código 6.6: Prática - Comandos de Repetição Aninhados

6.7 Coleção de Dados (Arrays)

6.7.1 Prática Proposta

Prática 6.7 — Coleção de Dados (Arrays). Elabore um algoritmo para calcular o salário líquido de 5 pessoas. Solicite ao usuário que digite seu nome e o valor de seu salário bruto. Peça também, para o usuário digitar a quantidade de dependentes. Para cada um dos dependentes deve ser solicitado o ganho mensal. Este valor deverá ser adicionado ao salário bruto. Calcule a renda familiar per capita. Caso a renda para cada membro da família seja menor que R\$ 500,00 a pessoa ficará isenta de imposto de renda, ou seja, não será calculado dentre as faixas salariais e seu valor será zero. Caso a renda para cada membro da família seja maior ou igual a R\$ 500,00 o imposto de renda a ser descontado do salário bruto deve considerar as seguintes regras:

- salário bruto (de 0,00 até 1.903,98): 5%;
- salário bruto (de 1.903,99 até 2.826,65): 7,5%;
- salário bruto (a partir de 2.826,66): 15%.

Ao final, mostre na tela o valor do salário líquido.

Armazene os salários líquidos em uma coleção indexada (array).

Depois, calcule a média de todos os salários líquidos. Posteriormente, mostre a média dos salários líquidos e quantas pessoas estão abaixo desse valor médio.

6.7.2 Gabarito da Prática



```
1 const prompt = require("prompt-sync")();
2
3 var salarios = [];
4 var soma = 0;
5
6 for(var i = 1; i <= 5; i++){
7     console.log("Pessoa ", i);
8
9     var nome = prompt("Digite seu nome: ");
```

```
10  var salario_bruto = parseFloat(prompt("Digite seu salário bruto: "));
11  var dependentes = parseInt(prompt("Digite o número de dependentes: ")
    );
12
13  for(var j = 1; j <= dependentes; j++){
14      console.log("Dependente ", j);
15      var ganho = parseFloat(prompt("Digite seu ganho mensal: "));
16      salario_bruto = salario_bruto + ganho;
17  }
18
19  var renda_percapta = salario_bruto / (dependentes + 1);
20
21  if (renda_percapta >= 500){
22
23      if (salario_bruto <= 1903.98){
24          var ir = salario_bruto * 0.05;
25      }else if(salario_bruto <= 2826.65){
26          var ir = salario_bruto * 0.075;
27      }else{
28          var ir = salario_bruto * 0.15;
29      }
30
31  }else{
32      var ir = 0;
33  }
34
35  var salario_liquido = salario_bruto - ir;
36  console.log("Salário líquido: R$", salario_liquido, "\n");
37
38  salarios.push(salario_liquido);
39  soma = soma + salario_liquido;
```

```
40 }
41
42 media = soma / 5;
43
44 var qtd_menores = 0;
45 for(var i = 0; i <= 4; i++){
46     if (salarios[i] < media){
47         qtd_menores++;
48     }
49 }
50
51 console.log("A média dos salários líquidos é: ", media);
52 console.log("A quantidade de pessoas com salário líquido menor que a média é: ", qtd_menores);
```

Exemplo de Código 6.7: Prática - Coleção de Dados (Arrays)

6.8 Funções e Procedimentos

6.8.1 Prática Proposta

Prática 6.8 — Funções e Procedimentos. Elabore um algoritmo para calcular o salário líquido de 5 pessoas. Solicite ao usuário que digite seu nome e o valor de seu salário bruto. Peça também, para o usuário digitar a quantidade de dependentes. Para cada um dos dependentes deve ser solicitado o ganho mensal. Este valor deverá ser adicionado ao salário bruto. Calcule a renda familiar per capita. Caso a renda para cada membro da família seja menor que R\$ 500,00 a pessoa ficará isenta de imposto de renda, ou seja, não será calculado dentre as faixas salariais e seu valor será zero. Caso a renda para cada membro da família seja maior ou igual a R\$ 500,00 o imposto de renda a ser descontado do salário bruto deve considerar as seguintes regras:

- salário bruto (de 0,00 até 1.903,98): 5%;
- salário bruto (de 1.903,99 até 2.826,65): 7,5%;
- salário bruto (a partir de 2.826,66): 15%.

Ao final, mostre na tela o valor do salário líquido.

Armazene os salários líquidos em uma coleção indexada (array).

Depois, calcule a média de todos os salários líquidos. Posteriormente, mostre a média dos salários líquidos e quantas pessoas estão abaixo desse valor médio.

Refatore seu código. Para isso, crie 2 funções, a saber:

- função para **calcular o IR**: esta função deve receber como parâmetros o salário bruto e a renda per capita, e, retornar ao final o valor do IR;
- função para **calcular a quantidade de salários líquidos menores que a média**: esta função deve receber como parâmetros o array de salários líquidos e a média dos salários líquidos, e, retornar a quantidade de salários líquidos menores que a média.

6.8.2 Gabarito da Prática



```
1 const prompt = require("prompt-sync")();
```

```
2
```

```
3 var salarios = [];  
4 var soma = 0;  
5  
6 for(var i = 1; i <= 5; i++){  
7     console.log("Pessoa ", i);  
8     var nome = prompt("Digite seu nome: ");  
9     var salario_bruto = parseFloat(prompt("Digite seu salário bruto: "));  
10    var dependentes = parseInt(prompt("Digite o número de dependentes: ")  
11        );  
12  
13    for(var j = 1; j <= dependentes; j++){  
14        console.log("Dependente ", j);  
15        var ganho = parseFloat(prompt("Digite seu ganho mensal: "));  
16        salario_bruto = salario_bruto + ganho;  
17    }  
18  
19    var renda_percapta = salario_bruto / (dependentes + 1);  
20  
21    var ir = calcular_ir(salario_bruto, renda_percapta);  
22  
23    var salario_liquido = salario_bruto - ir;  
24    console.log("Salário líquido: R$", salario_liquido, "\n");  
25  
26    salarios.push(salario_liquido);  
27    soma = soma + salario_liquido;  
28  
29 media = soma / 5;  
30  
31 qtd_menores = calcular_qtd_menores(salarios, media);  
32
```

```
33 console.log("A média dos salários líquidos é: ", media);
34 console.log("A quantidade de pessoas com salário líquido menor que a média é: ", qtd_menores);
35
36 function calcular_ir(salario_bruto, renda_percapta) {
37     if (renda_percapta >= 500){
38
39         if (salario_bruto <= 1903.98){
40             var ir = salario_bruto * 0.05;
41         }else if(salario_bruto <= 2826.65){
42             var ir = salario_bruto * 0.075;
43         }else{
44             var ir = salario_bruto * 0.15;
45         }
46
47     }else{
48         var ir = 0;
49     }
50     return ir;
51 }
52
53 function calcular_qtd_menores(salarios, media){
54     var qtd_menores = 0;
55     for(var i = 0; i <= 4; i++){
56         if (salarios[i] < media){
57             qtd_menores++;
58         }
59     }
60     return qtd_menores;
61 }
```

Exemplo de Código 6.8: Prática - Funções e Procedimentos