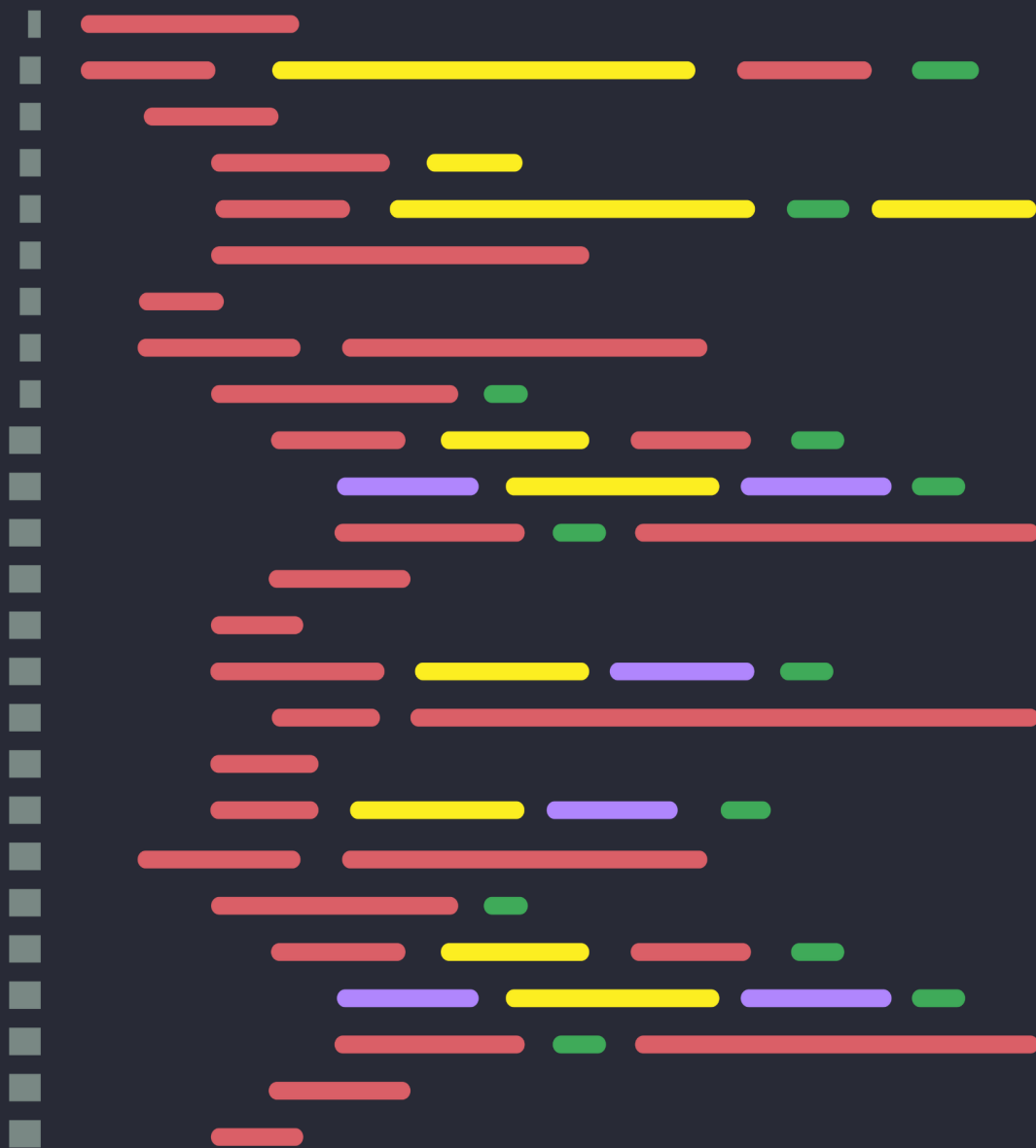


# Aprenda Programar com JavaScript



Jesiel Viana

# Aprenda Programar com JavaScript

Jesiel Viana

Esse livro está à venda em

<http://leanpub.com/aprenda-programar-com-javascript>

Essa versão foi publicada em 2024-03-21



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2024 Jesiel Viana

*Dedico este livro aos meus pais, cujo apoio incansável aos meus estudos foi fundamental para minha jornada. À minha esposa e minhas duas filhas, por serem minha fonte constante de inspiração e suporte diário. E também aos meus estimados alunos e aos colegas professores, cujo compartilhamento de conhecimento e colaboração enriquecem minha trajetória profissional. Este livro é uma expressão de gratidão a todos vocês.”*

# Conteúdo

<b>1. Introdução à Programação</b>	<b>1</b>
1.1. História da programação	2
1.2. Importância da programação no contexto atual	4
1.3. Conclusão	5
<b>2. Lógica de Programação e Algoritmos</b>	<b>6</b>
2.1. Lógica de programação	6
2.2. Algoritmos	7
2.3. Conclusão	12
2.4. Exercícios	13
<b>3. Introdução ao JavaScript</b>	<b>14</b>
3.1. Principais características	14
3.2. Surgimento e evolução	16
3.3. Ambientes de execução	17
3.4. Ferramentas para edição de código	19
3.5. Saída de dados com console.log()	22
3.6. Conclusão	23
3.7. Exercícios	24
<b>4. Tipos de Dados e Variáveis</b>	<b>25</b>
4.1. Tipos de Dados	25
4.2. Variáveis	27
4.3. Sintaxe básica do JavaScript	31
4.4. Entrada de dados com prompt()	34
4.5. Conversão de tipos de dados	35
4.6. Conclusão	38
4.7. Exercícios resolvidos	38
4.8. Exercícios	39
<b>5. Operadores e Estruturas de Controle de Fluxo</b>	<b>41</b>
5.1. Operadores	41
5.2. Controle de fluxo	48

5.3. Estruturas condicionais . . . . .	49
5.4. Laços de repetição . . . . .	57
5.5. Conclusão . . . . .	62
5.6. Exercícios resolvidos . . . . .	63
5.7. Exercícios . . . . .	65
<b>6. Funções e Escopo . . . . .</b>	<b>67</b>
6.1. Funções . . . . .	67
6.2. Funções predefinidas . . . . .	67
6.3. Declaração de funções . . . . .	69
6.4. Execução de funções . . . . .	73
6.5. Escopo . . . . .	76
6.6. Conclusão . . . . .	78
6.7. Exercícios . . . . .	78
<b>7. Estruturas de dados básicas . . . . .</b>	<b>81</b>
7.1. Estruturas de dados . . . . .	81
7.2. Arrays . . . . .	81
<b>8. Modularização . . . . .</b>	<b>86</b>
<b>9. JavaScript Orientado a Objetos . . . . .</b>	<b>87</b>
<b>10. Testes Automatizados . . . . .</b>	<b>88</b>
<b>11. Introdução a TypeScript . . . . .</b>	<b>89</b>

# 1. Introdução à Programação

A programação de computadores é a prática de criar sequências de instruções, chamadas de código, para que um computador execute tarefas específicas. Essas instruções são escritas em linguagens de programação compreensíveis para os desenvolvedores e, posteriormente, traduzidas para linguagem de máquina, que o computador pode executar diretamente.

A programação é uma habilidade essencial no desenvolvimento de software, permitindo que os programadores criem aplicativos, websites, jogos e outras soluções digitais. Envolve a aplicação de lógica, algoritmos e a compreensão das capacidades da máquina para resolver problemas e automatizar tarefas.

Aprender a programar vai além de apenas criar softwares. É um processo que desenvolve o seu raciocínio lógico, a sua capacidade de resolver problemas e a sua criatividade. Ao programar, você se torna um pensador computacional, capaz de analisar situações e encontrar soluções inovadoras.

Para iniciar a jornada na programação, é imprescindível adquirir conhecimento em uma linguagem de programação. Assim como falamos diferentes idiomas para nos comunicarmos com pessoas de culturas variadas, as linguagens de programação são como idiomas usados para nos comunicarmos com computadores.

Linguagens de programação são ferramentas desenvolvidas pelos seres humanos para transmitir instruções de maneira compreensível para os computadores. Sua concepção visa se aproximar da linguagem natural que utilizamos ao falar e escrever, o que torna mais intuitivo para os programadores expressarem suas intenções e lógica ao desenvolver software. Essa abordagem facilita a criação de um ambiente onde a interação entre humanos e máquinas se dá de maneira eficiente e precisa.

Algumas das linguagens de programação mais utilizadas no mercado profissional são:

- **Java:** Linguagem orientada a objetos, reconhecida por sua portabilidade e robustez, tornando-a especialmente adequada para ambientes de desenvolvimento corporativo;

- **Python:** Linguagem conhecida por sua sintaxe clara e concisa. É extensivamente empregada na área de ciência de dados;
- **JavaScript:** Reconhecida como a linguagem mais popular do mundo ([GITHUB, 2024<sup>1</sup>](https://github.blog/2023-11-08-the-state-of-open-source-and-ai)), desempenha um papel crucial no desenvolvimento web moderno. Reconhecida por sua versatilidade, permite a criação não apenas de aplicações web, mas também de aplicativos móveis e desktop.

Ao longo deste livro, exploraremos os fundamentos da programação utilizando o JavaScript como principal ferramenta, proporcionando uma experiência prática e relevante para quem busca iniciar sua jornada na programação. A escolha do JavaScript se justifica por sua popularidade, facilidade de uso sem a necessidade de configurações complexas, versatilidade e influência predominante no desenvolvimento web contemporâneo.

## 1.1. História da programação

A história da programação de computadores é uma jornada fascinante que começou no século XIX e se desenvolveu significativamente ao longo do século XX.

Estudar as origens da programação é fundamental para entender como a tecnologia evoluiu e como os softwares que utilizamos hoje foram criados. Aqui estão alguns marcos importantes:

- **Século XIX (1800):** Ada Lovelace é reconhecida como a primeira programadora ao escrever o primeiro algoritmo para ser processado por uma máquina, a máquina analítica de Charles Babbage. Essa máquina era capaz de realizar cálculos complexos e até mesmo escrever programas simples, utilizando cartões perfurados para armazenar instruções.
- **Décadas de 1930 e 1940:** Alan Turing contribui com a Máquina de Turing, estabelecendo fundamentos teóricos para a computação. Nesse mesmo período, durante a Segunda Guerra Mundial, foi construído o ENIAC (Electronic Numerical Integrator and Computer), primeiro computador digital eletrônico de grande escala.
- **Anos 1950:** Surgimento das primeiras linguagens de programação de alto nível, como Fortran e COBOL. Grace Hopper desenvolve o primeiro compilador.

---

<sup>1</sup><https://github.blog/2023-11-08-the-state-of-open-source-and-ai>

- **Década de 1970:** Nascimento das linguagens de alto nível: C, Pascal e Smalltalk surgem, facilitando a escrita de programas e abrindo caminho para a popularização da programação. Ascensão da programação orientada a objetos com Smalltalk. Desenvolvimento do microcomputador: O Altair 8800 e o Apple II trazem a computação para as casas, impulsionando o interesse pela programação entre o público em geral.
- **Década de 1980:** A explosão dos computadores pessoais: O IBM PC e o Macintosh democratizam o acesso à tecnologia e criam um mercado em massa para softwares. Evolução das linguagens de programação: C++ e Objective-C surgem como linguagens híbridas que combinam características de linguagens anteriores. Surgimento de interfaces gráficas amigáveis, como a do Windows e do Mac OS, torna os computadores mais intuitivos e acessíveis.
- **Década de 1990:** Nascimento da Web, criada por Tim Berners-Lee, revolucionou a forma como as pessoas se comunicavam e acessavam informações. Crescimento exponencial da internet. Desenvolvimento de navegadores web. Criação de novas linguagens de programação: Linguagens como Java, JavaScript e Python foram criadas nesse período para atender às demandas específicas da web. Popularização dos jogos online e crescimento do comércio eletrônico.
- **Década de 2000 em diante:** Ascensão das redes sociais, popularização de smartphones e o desenvolvimento de técnicas de inteligência artificial, que revolucionou diversos setores da sociedade e impulsionou a criação de softwares cada vez mais inteligentes.

Embora as tecnologias do século XIX não fossem computadores em si, elas estabeleceram princípios fundamentais para o desenvolvimento da programação moderna. As ideias de automação, lógica computacional e armazenamento de instruções em cartões perfurados abriram caminho para a criação dos primeiros computadores e linguagens de programação no século XX.

Embora a programação ainda estivesse em seus primórdios nas décadas de 1930 e 1940, as inovações desse período lançaram as bases para o avanço da tecnologia proporcionando inovações posteriores. Os primeiros computadores e linguagens de programação abriram caminho para a criação de softwares que transformaram a maneira como vivemos, trabalhamos e nos divertimos.

As inovações das décadas de 1950 a 1980 foram fundamentais para o desenvolvimento da programação moderna. As inovações desse período lançaram as bases para a criação de softwares cada vez mais complexos e poderosos. As



linguagens, ferramentas e conceitos desenvolvidos nesse período continuam a ser usados até hoje, influenciando a maneira como criamos e utilizamos softwares em diversos campos.

A década de 1990 foi um período de grande transformação para o mundo da programação. A internet e a web abriram um novo mundo de possibilidades para os programadores, que desenvolveram softwares inovadores que mudaram a maneira como vivemos, trabalhamos e nos divertimos.

O período de 2000 em diante foi marcado por uma aceleração no ritmo das inovações tecnológicas. A programação se tornou cada vez mais essencial para diversos setores da sociedade, e os programadores passaram a ter um papel fundamental no desenvolvimento de soluções para os desafios do mundo moderno.

## 1.2. Importância da programação no contexto atual

Aprender programação de computadores é uma habilidade valiosa e oferece uma série de benefícios tanto no contexto profissional quanto pessoal, tendo em vista que a programação de computadores desempenha um papel fundamental no contexto atual por diversas razões. Aqui estão algumas razões importantes para aprender programação:

- **Desenvolvimento de software:** A programação é essencial para a criação de aplicativos e softwares que atendem às necessidades variadas dos usuários, desde aplicativos móveis até sistemas complexos de gerenciamento empresarial.
- **Inovação tecnológica:** A programação impulsiona a inovação, possibilitando o desenvolvimento de novas tecnologias, softwares e soluções que melhoram a eficiência, a comunicação e a qualidade de vida.
- **Transformação digital:** Em um mundo cada vez mais digital, a programação é a espinha dorsal da transformação digital, permitindo que empresas e organizações migrem para plataformas online, automatizem processos e ofereçam serviços digitais.
- **Conectividade e Internet das Coisas (IoT):** A programação facilita a interconexão de dispositivos e a criação de sistemas que compõem a Internet das Coisas, possibilitando a comunicação entre objetos e coleta de dados em tempo real.

- **Ciência de dados:** Na era da informação, a programação é crucial para lidar com grandes conjuntos de dados, desenvolver algoritmos de análise e criar modelos preditivos que sustentam a ciência de dados.
- **Inteligência artificial:** A programação é a base para o desenvolvimento de sistemas de inteligência artificial e aprendizado de máquina, possibilitando a criação de algoritmos capazes de aprender e tomar decisões autônomas.
- **Empregabilidade:** Há uma demanda por desenvolvedores de software em diversos setores. Aprender programação aumenta suas oportunidades de emprego.

Além disso, aprender a programar conecta você a uma comunidade global de desenvolvedores, proporcionando oportunidades de aprendizado e networking.

## 1.3. Conclusão

Em resumo, a programação é uma habilidade versátil e poderosa que permeia todos os aspectos da sociedade moderna, impulsionando a inovação, a eficiência e o progresso tecnológico. Aprender programação é uma jornada recompensadora que não apenas abre portas para diversas oportunidades profissionais, mas também promove o desenvolvimento de habilidades cognitivas e criativas essenciais para enfrentar os desafios do mundo atual.

## 2. Lógica de Programação e Algoritmos

Entender a lógica é fundamental para quem quer aprender a programar. Antes de começarmos a escrever código, é importante compreender como pensar de maneira organizada para resolver problemas. A lógica nos ensina a seguir passos lógicos para resolver desafios complexos. Em resumo, lógica é a arte de pensar de forma organizada e estruturada.

### 2.1. Lógica de programação

Lógica de programação é a capacidade de criar sequências lógicas de instruções para que um computador realize uma tarefa específica de forma eficaz.

A lógica de programação não está atrelada a uma linguagem de programação específica, mas sim ao raciocínio lógico que precede a codificação.

Principais aspectos da lógica de programação:

- **Sequencialidade:** A lógica de programação baseia-se na execução sequencial de instruções. Cada passo é executado em ordem, do início ao fim, formando uma sequência lógica.
- **Condicionais:** Estruturas condicionais permitem que o programa escolha entre diferentes caminhos de execução com base em condições lógicas.
- **Repetição:** Estruturas de repetição possibilitam a execução repetida de um bloco de código, facilitando a manipulação de conjuntos de dados ou a resolução de problemas iterativos.
- **Abstração e modularidade:** A habilidade de abstrair problemas complexos, dividindo-os em partes menores e mais gerenciáveis facilita a compreensão e manutenção do código.
- **Resolução de problemas:** A lógica de programação é, em sua essência, a resolução de problemas utilizando algoritmos. Compreender um problema,

identificar entradas e saídas esperadas, e criar uma sequência lógica de passos para chegar à solução são competências centrais.

Lógica de programação e algoritmos são fundamentais para resolver problemas por meio da programação. Ao dominá-los, torna-se mais fácil aprender linguagens de programação e desenvolver habilidades para pensar logicamente, organizar etapas e resolver problemas em diferentes situações. A seguir, serão apresentados o conceito e os tipos de representação de algoritmos.

## 2.2. Algoritmos

Algoritmo é uma sequência de passos precisos projetados para realizar uma tarefa específica. Em outras palavras, é um conjunto de passos lógicos e específicos que devem ser seguidos para alcançar um objetivo ou resolver um determinado problema.

No contexto da programação de computadores, um algoritmo é como uma receita que orienta o computador a executar uma tarefa específica. Ele estabelece uma sequência precisa de instruções que o computador deve seguir para atingir o resultado desejado.

Um algoritmo possui um conjunto de características bem definidas, que são:

- Início
- Fim
- Não ambíguo
- Capacidade de receber dados de entrada
- Possibilidade de gerar informações de saída
- Finito

O processo de escrita de algoritmos inclui três etapas:

1. Interpretar o problema: analisar e entender o problema para identificar uma provável solução;
2. Identificar entradas e saídas;

3. Determinar a sequência de passos: o que deve ser feito para transformar a entrada em uma saída válida.

Para iniciantes em programação, antes de desenvolver algoritmos em uma linguagem de programação real, é importante compreender suas etapas de construção. Existem diversas formas de descrever e visualizar algoritmos de maneira eficaz, incluindo a narrativa, o pseudocódigo e os fluxogramas. Essas representações são fundamentais para auxiliar na compreensão da execução de algoritmos. A seguir, detalharemos cada uma delas.

## 2.2.1. Narrativa

A forma narrativa de representação de algoritmos é uma abordagem textual para descrever passo a passo a sequência de operações necessárias para resolver um problema específico. Nesse formato, o algoritmo é expresso em linguagem natural, utilizando frases e parágrafos para descrever cada etapa do processo.

*A representação **narrativa** é a minha preferida, pois acredito que é mais acessível para quem não tem familiaridade com programação, uma vez que utiliza linguagem natural e se concentra exclusivamente na lógica do problema.*

Ao empregar a abordagem narrativa, o autor do algoritmo descreve cada ação com termos acessíveis a qualquer pessoa, mesmo aquelas sem conhecimento prévio de linguagens de programação.

A seguir será apresentado um exemplo de representação narrativa de um algoritmo para calcular a média de dois números:

Figura 2.1. Algoritmo em formato narrativo

- 
- 1 Início do algoritmo.
  - 2 Solicitar ao usuário que forneça o primeiro número.
  - 3 Armazenar o primeiro número.
  - 4 Solicitar ao usuário que forneça o segundo número.
  - 5 Armazenar o segundo número.
  - 6 Calcular a média dos dois números somando ambos, e dividindo por 2.
  - 7 Armazenar o resultado da média.
  - 8 Exibir a média calculada ao usuário.
  - 9 Fim do algoritmo.
-

Essa representação descreve passo a passo as ações a serem executadas, proporcionando uma compreensão clara do fluxo lógico do algoritmo, sem a necessidade de preocupações com a sintaxe de uma linguagem de programação específica.

Outra abordagem frequentemente utilizada é conhecida como pseudocódigos, que são representações intermediárias entre a linguagem natural e a linguagem de programação.

### 2.2.2. Pseudocódigo

Pseudocódigo é uma abordagem simplificada para escrever algoritmos, utilizando uma linguagem próxima à linguagem natural do autor. Funciona como uma linguagem intermediária entre o idioma nativo do programador e a linguagem que os computadores entendem, com o uso de convenções que facilitam a transição para uma linguagem de programação específica.

Entre as linguagens de pseudocódigo utilizadas para ensino e prática de programação, destaca-se o Portugol como uma das opções mais populares no Brasil. Criada no Brasil na década de 1980 para fins educacionais, essa pseudolinguagem auxilia no aprendizado de conceitos básicos de algoritmos e lógica de programação, utilizando uma sintaxe próxima do português. Geralmente adotada como uma etapa inicial antes do estudo de linguagens de programação reais.

Principais características do Portugol:

- **Sintaxe simples:** A sintaxe do Portugol é simplificada e próxima à linguagem natural, facilitando o entendimento para quem está começando a aprender programação.
- **Palavras-chave em Português:** Utiliza palavras-chave em português, o que torna o código mais acessível para falantes nativos dessa língua.
- **Ambiente de desenvolvimento:** Simples e acessível, especialmente para iniciantes em programação. Geralmente, oferece uma interface intuitiva e amigável, com recursos básicos para escrever, testar e depurar algoritmos.

A seguir, observe um exemplo de pseudocódigo em Portugol para calcular a média de dois números. As linhas que iniciam com “//” são comentários de código, utilizadas para descrever o código, essas linhas são ignoradas pelo programa.

Figura 2.2. Algoritmo em Portugol

---

```
1  variaveis
2      // Declaração de Variáveis
3      num1, num2, media: real;
4
5  inicio
6      // Passo 1: Solicitar e armazenar o primeiro número
7      escrever "Digite o primeiro número: ";
8      ler num1;
9
10     // Passo 2: Solicitar e armazenar o segundo número
11     escrever "Digite o segundo número: ";
12     ler num2;
13
14     // Passo 3: Calcular a média
15     media <- (num1 + num2) / 2;
16
17     // Passo 4: Exibir a média
18     escrever "A média dos números é: ", media;
19 fim
```

---

Neste pseudocódigo em Portugol, utiliza-se a sintaxe característica dessa linguagem para representar o algoritmo de cálculo da média de dois números. As palavras-chave como “variaveis,” “inicio,” “escrever,” e “ler” são específicas do Portugol e auxiliam na compreensão da lógica do programa.







Eu particularmente não gosto de usar Portugol para ensinar programação, ao invés de investir tempo para aprender uma sintaxe de uma pseudolinguagem eu recomendo ir direto para uma linguagem de programação real. Quando preciso representar algoritmos antes de implementá-los em uma linguagem de programação real, costumo utilizar fluxogramas e, principalmente, descrição narrativa do algoritmo.

### 2.2.3. Fluxogramas

Fluxograma é uma representação visual de um processo ou algoritmo, utilizando símbolos gráficos interconectados. Ele descreve a sequência de passos necessários para realizar uma determinada tarefa ou resolver um problema, ajudando a visualizar e compreender a lógica por trás do processo.

Os elementos gráficos comuns em um fluxograma incluem formas geométricas, como retângulos, losangos e círculos, que representam diferentes etapas do processo, e setas que indicam a direção do fluxo de controle entre essas etapas. A Tabela 1 apresenta a simbologia de fluxograma, normatizada pela ISO 5807, utilizada para descrever algoritmos.

Figura 2.3. Simbologia de fluxograma normatizada pela ISO 5807

Símbolo	Significado	Descrição
	Início/Fim	Indica o início ou o término do algoritmo
	Entrada/Saída	Reflete a entrada ou saída de dados
	Processamento	Representa uma operação ou ação a ser realizada
	Decisão	Apresenta uma condição lógica que direciona o fluxo
	Exibição	Representa a exibição visual do resultado
	Conector	Indica a direção do fluxo entre etapas

Usando a simbologia acima, a Figura abaixo representa o fluxograma do algoritmo para calcular a média de dois números.



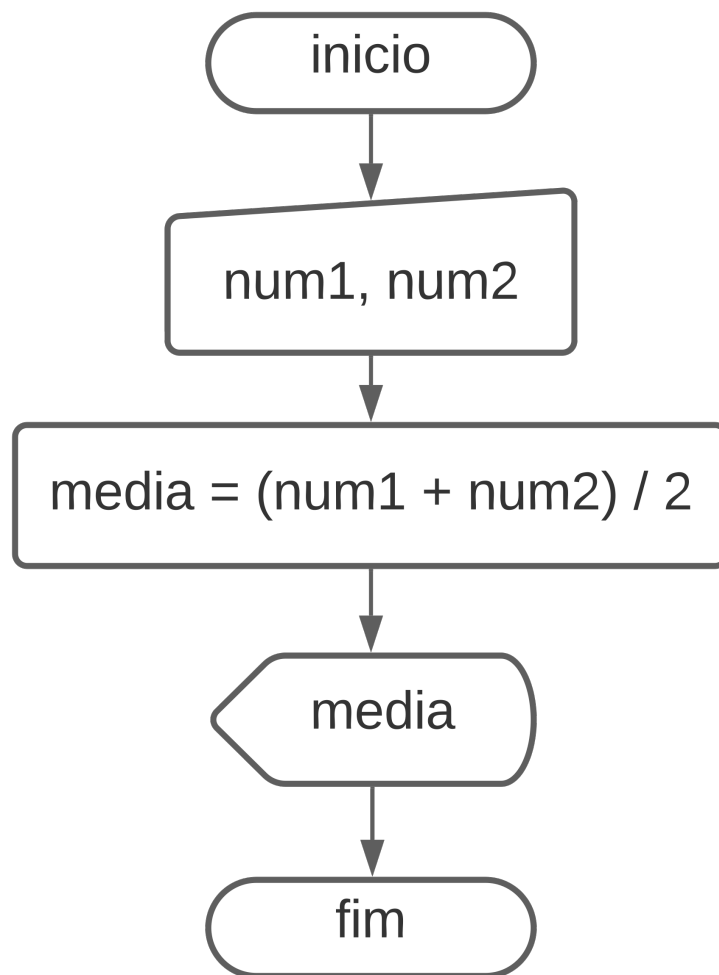


Figura 2.4. Fluxograma do algoritmo que calcula a média de dois números

Construir fluxogramas eficazes requer uma compreensão clara do algoritmo. É necessário identificar o ponto de início e, em seguida, descrever cada passo de forma sequencial, utilizando símbolos apropriados para representar operações específicas e conectando-os logicamente para formar o fluxo coerente do algoritmo.

## 2.3. Conclusão

A lógica de programação, como pensamento abstrato e conceitual, antecede a criação de algoritmos, os quais representam a materialização prática dessa lógica, consistindo em sequências concretas de instruções para resolver problemas ou executar tarefas específicas. Enquanto a lógica de programação representa a base conceitual, os algoritmos são sua aplicação prática no campo da programação de computadores.

Quanto à descrição de algoritmos, a escolha da melhor abordagem depende do contexto e do público-alvo. Para iniciantes na área da programação, a abordagem narrativa se destaca, pois possibilita uma explicação passo a passo em linguagem natural, facilitando a compreensão e oferecendo uma visão detalhada da lógica subjacente aos algoritmos. Contudo, o conhecimento de diversas ferramentas de representação nos permite explorar os algoritmos com mais confiança e criatividade.

**Lembre-se:** Aprender lógica de programação é um processo que exige tempo e dedicação. Comece com passos pequenos, pratique com frequência e não desista! A recompensa será a capacidade de criar seus próprios programas e ferramentas, abrindo portas para diversas áreas da tecnologia.

## 2.4. Exercícios

Esses exercícios abordam conceitos fundamentais de lógica de programação e algoritmos e são ótimos para praticar e consolidar o conhecimento adquirido. Para responder as questões abaixo utilize algumas das formas de representação de algoritmos descritas neste capítulo.

1. Escreva um algoritmo para calcular a média de três números.
2. Crie um algoritmo que determine se um número é par ou ímpar.
3. Crie um algoritmo que calcule o fatorial de um número.
4. Desenvolva um algoritmo para verificar se uma palavra é um palíndromo.
5. Crie um algoritmo para encontrar a soma dos dígitos de um número inteiro.

# 3. Introdução ao JavaScript

Bem-vindo ao mundo do JavaScript! Neste capítulo introdutório, mergulharemos no fascinante universo da linguagem de programação JavaScript. Como uma das linguagens mais populares e versáteis da atualidade, o JavaScript oferece uma entrada acessível para o vasto mundo da programação.

Começaremos explorando a natureza das linguagens de programação e como elas servem como ponte entre a mente humana e a máquina. Veremos como o JavaScript, em particular, se destaca por sua sintaxe simplificada e flexível, tornando-a uma excelente escolha para iniciantes.

Linguagens de programação são ferramentas criadas pelos desenvolvedores para comunicar instruções de forma compreensível para os computadores. Elas são projetadas para se aproximar da linguagem natural que usamos ao falar e escrever, tornando mais intuitivo para os programadores expressarem suas intenções e lógica ao criar software. Entre as linguagens de programação mais utilizadas na atualidade, destacam-se C, C#, Python, Java e JavaScript.

Entre as várias linguagens de programação, o JavaScript é minha escolha para iniciantes. Sua sintaxe é simples e fácil de entender, tornando o processo de aprendizado mais suave para quem está começando. Além disso, o JavaScript tem a vantagem de poder ser executado diretamente em qualquer navegador web, sem a necessidade de configurações complicadas, o que facilita ainda mais a familiarização com a linguagem.

Como o JavaScript é amplamente utilizado no desenvolvimento web, os iniciantes podem ver rapidamente os resultados do seu trabalho ao criar páginas web interativas e dinâmicas, além de jogos e aplicativos móveis. Ademais, há uma grande quantidade de recursos educacionais disponíveis online e uma comunidade ativa pronta para ajudar, o que faz do JavaScript uma escolha acolhedora e inspiradora para quem está dando os primeiros passos na programação.

Em suma, JavaScript é uma escolha ideal para iniciantes devido à sua acessibilidade, versatilidade, poder e curva de aprendizado gradual. Se você está considerando aprender a programar, JavaScript é uma excelente opção para iniciar sua jornada. Venha comigo nessa!

## 3.1. Principais características

O JavaScript possui várias características distintivas que o tornam uma linguagem de programação amplamente utilizada e versátil. Aqui estão algumas das principais características do JavaScript:

- **Sintaxe simplificada e flexível:** JavaScript possui uma sintaxe simples e flexível, o que a torna relativamente fácil de aprender e usar. Sua sintaxe é semelhante à de outras linguagens de programação como Java e C, mas com menos complexidade.
- **Tipagem dinâmica:** Não há necessidade de declarar o tipo de cada variável, facilitando a compreensão para iniciantes. Permite que o foco seja direcionado aos conceitos fundamentais da programação, sem a necessidade de se preocupar com tipos de dados específicos.
- **Case Sensitivity:** JavaScript é sensível a maiúsculas e minúsculas, ou seja, nome e Nome são tratados como duas coisas diferentes.
- **Interpretado:** Não precisa ser compilado antes da execução, possibilitando um desenvolvimento mais rápido e dinâmico. Permite a depuração de erros de forma mais fácil e eficiente.
- **Suporte para programação orientada a objetos:** JavaScript oferece suporte à programação orientada a objetos (POO), permitindo que os desenvolvedores utilizem classes, objetos, herança e outros mecanismos para organizar, estruturar e reutilizar o código de maneira eficiente.
- **Ecossistema:** JavaScript possui uma vasta coleção de bibliotecas e frameworks que facilitam o desenvolvimento de aplicações complexas e poderosas.
- **Multiplataforma:** JavaScript é uma linguagem multiplataforma, o que significa que pode ser executada em diferentes sistemas operacionais e dispositivos.
- **Código aberto:** Com uma especificação aberta e várias implementações gratuitas de código aberto disponíveis, qualquer pessoa pode contribuir para o desenvolvimento do JavaScript e usá-lo livremente.
- **Alto desempenho:** Código JavaScript é executado de forma rápida e eficiente pelos navegadores modernos e plataformas de servidor como Node.js e Bun.
- **Versatilidade:** JavaScript pode ser usado para criar diversos tipos de aplicações, desde websites interativos até jogos e aplicativos móveis.

- **Demanda no mercado:** A demanda por desenvolvedores JavaScript é alta no mercado de trabalho, o que significa que aprender a linguagem pode te abrir para diversas oportunidades. Segundo o [GitHub](https://octoverse.github.com/)<sup>1</sup> e [StackOverflow](https://survey.stackoverflow.co/)<sup>2</sup>, é a linguagem mais utilizada no mundo.

Com tantas características e vantagens, JavaScript se consolida como uma das linguagens de programação mais populares e utilizadas no mundo. Em resumo, JavaScript é uma linguagem poderosa, versátil e fácil de aprender, com um ecossistema rico e uma comunidade vibrante. É uma excelente escolha para iniciantes e experientes que desejam criar aplicações web e multiplataforma interativas e dinâmicas.

## 3.2. Surgimento e evolução

JavaScript teve sua origem em meados da década de 1990. Sua criação é creditada a Brendan Eich, enquanto trabalhava na *Netscape Communications*, empresa pioneira na era da internet. Abaixo será listado os principais acontecimentos na história do JavaScript.

- **1995:** O JavaScript é criado pela Netscape Communications Corporation, liderada por Brendan Eich. Inicialmente, é chamado de LiveScript, mas logo é renomeado para JavaScript.
- **1996:** O JavaScript é submetido à Ecma International para padronização, resultando na criação do padrão *ECMAScript*.
- **1997:** A primeira versão do *ECMAScript* (*ES1*) é lançada como um padrão oficial.
- **2005:** O JavaScript ganha destaque com o lançamento do *AJAX* (*Asynchronous JavaScript and XML*), uma técnica que permite que as páginas web façam solicitações assíncronas ao servidor, resultando em páginas mais dinâmicas e interativas.
- **2009:** O surgimento do Node.js foi um marco significativo, permitindo a execução do JavaScript no lado do servidor e abrindo portas para o desenvolvimento de aplicações web completas com JavaScript.

---

<sup>1</sup><https://octoverse.github.com/>

<sup>2</sup><https://survey.stackoverflow.co/>

- **2012-presente:** A criação de frameworks modernos como *React*, *Angular* e *Vue.js* simplificou ainda mais o desenvolvimento de interfaces web complexas e de alta performance, consolidando o JavaScript como uma das linguagens de programação mais utilizadas no mundo.
- **2015:** Um grande marco na história do JavaScript é alcançado com o lançamento do ECMAScript 6 (ES6), que introduz uma série de novos recursos importantes, como classes, *arrow functions*, *let* e *const*, e muito mais.

Esses são apenas alguns dos principais acontecimentos na história do JavaScript. Ao longo dos anos, o JavaScript se tornou uma das linguagens de programação mais populares e amplamente utilizadas, e continua a evoluir e se adaptar às necessidades dos desenvolvedores e da web moderna.

JavaScript e ECMAScript são termos que frequentemente são usados de forma intercambiável, mas eles têm significados ligeiramente diferentes. Em resumo, JavaScript é uma linguagem de programação específica que implementa o padrão ECMAScript, e o ECMAScript é o padrão que define a linguagem de programação JavaScript e suas características.

### 3.3. Ambientes de execução

Como mencionado anteriormente, JavaScript é uma linguagem de programação versátil que pode ser executada em diversas plataformas.

O JavaScript foi inicialmente desenvolvido para funcionar em navegadores da web, permitindo a criação de páginas interativas e dinâmicas na Internet. Com o tempo, surgiram ambientes independentes dos navegadores para a execução do JavaScript, como [Node.js](https://nodejs.org)<sup>3</sup>, [Deno](https://deno.com)<sup>4</sup> e [Bun](https://bun.sh)<sup>5</sup>, conhecidos como plataformas de execução de JavaScript do lado do servidor. O Node.js foi pioneiro e é amplamente utilizado.

Neste livro, optamos por utilizar o Bun como plataforma principal para executar JavaScript fora dos navegadores. O Bun é uma plataforma moderna, fácil de instalar e usar, sendo especialmente vantajoso para iniciantes devido à ausência de configurações iniciais. No entanto, é importante ressaltar que qualquer código JavaScript válido pode ser executado em qualquer ambiente, geralmente sem a necessidade de modificações.

---

<sup>3</sup><https://nodejs.org>

<sup>4</sup><https://deno.com>

<sup>5</sup><https://bun.sh>

### 3.3.1. Navegador Web

Para executar código JavaScript diretamente no navegador, você só precisa de um navegador web, como o Google Chrome, Mozilla Firefox ou qualquer outro navegador moderno.

Basta abrir o navegador, acessar o *console* de desenvolvedor pressionando F12 ou clicando com o botão direito do mouse na página e selecionando “Inspecionar” ou “Console”, e então você pode começar a escrever e executar código JavaScript no *console*.

Não há necessidade de instalar ou configurar nada adicionalmente, já que os navegadores modernos já possuem um mecanismo JavaScript embutido. **Utilize o console do navegador somente para testar e depurar seu código.**

### 3.3.2. Bun

Para executar código JavaScript no Bun, é necessário instalá-lo primeiro em seu computador. O Bun proporciona uma experiência de desenvolvimento mais simples e intuitiva do que o Node.js, com menos configurações e um tempo de inicialização mais rápido. Além disso, ele oferece mais recursos nativos do que o Node.js, mantendo uma excelente compatibilidade entre eles, assim como com os navegadores. Isso significa que o mesmo código JavaScript que você executa no Bun pode ser executado no Node.js ou em um navegador sem a necessidade de fazer alterações.

Para instalar o Bun no macOS e Linux, basta executar o seguinte comando no terminal do seu computador:

```
curl -fsSL https://bun.sh/install | bash
```

Para Windows ainda não há uma versão estável, mas pode-se instalar uma versão experimental digitando o seguinte comando no *PowerShell/cmd.exe*:

```
powershell -c "irm bun.sh/install.ps1|iex"
```

Depois que o comando acima for processado, verifique se a instalação foi bem-sucedida digitando no terminal/PowerShell/cmd:

```
bun -v
```

Isso deverá exibir a versão instalada do Bun. Se tiver alguma dúvida, você pode acessar o site oficial do Bun em <https://bun.sh/> e seguir as instruções de instalação para o seu sistema operacional.

Com o Bun instalado, você pode executar arquivos JavaScript pelo terminal do seu computador ou diretamente pelo terminal integrado da sua ferramenta de codificação, o que é mais prático. Basta digitar no terminal o comando `bun nomeArquivo.js`.

Com a instalação simples do Bun e sem a necessidade de configurações extras, os desenvolvedores têm à disposição um ambiente ágil e eficiente para testar e depurar seus scripts JavaScript.

## 3.4. Ferramentas para edição de código

Ao iniciar em JavaScript, escolha entre o editor de código *Visual Studio Code* (VS Code), que oferece muitos recursos e uso offline, ou a plataforma online *Replit*, ideal para iniciantes sem necessidade de instalação.

### 3.4.1. *Replit* Online

Replit é uma plataforma online que facilita a edição e execução de código em várias linguagens de programação, incluindo JavaScript, sem a necessidade de instalações no seu computador. É uma ótima ferramenta para iniciantes em programação, pois possibilita que eles se concentrem exclusivamente no aprendizado da lógica e sintaxe da linguagem, sem se preocupar com instalações e configurações adicionais.

Com uma interface intuitiva e amigável, o ambiente de desenvolvimento oferecido pelo Replit é fácil de usar, tornando a experiência de programação mais acessível. Além disso, destaca-se pela capacidade de colaboração em tempo real com outros desenvolvedores, proporcionando um ambiente dinâmico e colaborativo para o aprendizado.

Além disso, o Replit disponibiliza aplicativos nativos para sistemas operacionais como Windows, Linux e Mac, além de dispositivos móveis com Android e iOS. O aplicativo para dispositivos móveis permite aos usuários programarem diretamente de seus celulares, proporcionando flexibilidade e acessibilidade



para pessoas que não possuem computadores, possibilitando que elas também participem do mundo da programação, proporcionando uma experiência de programação fluida e acessível via dispositivo móvel.

Abaixo será apresentado um tutorial de como criar um projeto com JavaScript e Bun no Replit:

1. **Acessar ao Replit:** Acesse o site do Replit em [replit.com](https://replit.com)<sup>6</sup> e faça login em sua conta ou crie uma nova, caso ainda não tenha uma.
2. **Criar um novo projeto a partir do template do Bun JS:** Para começar um novo projeto Bun JS, siga estes passos: clique no link do template (<https://replit.com/@Jesielviana/Bun-JS>) e, em seguida, pressione o botão verde “Use Template”. Uma janela modal será aberta, solicitando que você insira o nome e a descrição do projeto. Depois de preenchidos, clique novamente em “Use Template”, localizado na parte inferior direita da janela.
3. **Estrutura do Projeto:** Ao seguir os passos acima, você terá criado um projeto com um arquivo `index.js`. Escreva seu código JavaScript neste arquivo para ser executado pelo Bun.
4. **Executar o Projeto:** Para executar o projeto clique no botão verde “Run” localizado no topo da tela. A saída da execução do código será exibida no console do Replit, acessível em uma aba à direita da tela.
5. **Compartilhar o projeto:** Assim como em qualquer projeto Replit, você pode compartilhar o link direto para o seu projeto Bun ou convidar outras pessoas para colaborar diretamente nele, clicando no botão “Invite” no canto superior direito da tela.
6. **Salvar o Projeto:** O Replit salva automaticamente o seu projeto conforme você faz alterações no código.

Seguindo esses passos simples, você será capaz de criar e executar projetos JavaScript facilmente utilizando o *Replit*.

Após a criação do projeto, você pode fechar a aba do Replit ou fazer logout da sua conta a qualquer momento. Seu projeto será automaticamente salvo e estará disponível para acesso posterior em qualquer dispositivo conectado à internet.

---

<sup>6</sup><https://replit.com/>

### 3.4.2. VS Code

O VS Code é um editor de código leve, poderoso e altamente personalizável, desenvolvido pela Microsoft, que oferece uma experiência intuitiva e amigável para programadores de todos os níveis de habilidade. É uma ferramenta poderosa e versátil que pode ser utilizada para programar em diversas linguagens, incluindo JavaScript, Python, Java, C++, HTML, CSS e muitas outras.

O VS Code é mais do que um simples editor de código. É um Ambiente de Desenvolvimento Integrado (IDE) que oferece tudo o que você precisa para programar com eficiência e produtividade.

IDE significa Ambiente de Desenvolvimento Integrado (do inglês *Integrated Development Environment*). É uma ferramenta que combina diferentes recursos para auxiliar os desenvolvedores na criação de software.

Algumas características do VS Code que o tornam como uma das principais escolhas por desenvolvedores são:

- **Gratuito e Open Source:** O VS Code é totalmente gratuito e *open source*;
- **Multiplataforma:** O VS Code é compatível com Windows, macOS e Linux, permitindo que você o utilize em qualquer sistema operacional.
- **Leve e Rápido:** O VS Code é um editor de código leve e rápido, garantindo que ele não sobrecarregue seu computador durante o uso.
- **Interface Personalizável:** A interface do VS Code é altamente personalizável, possibilitando que você a ajuste de acordo com suas preferências e necessidades.
- **Extensões:** O VS Code possui uma grande variedade de extensões que podem ser utilizadas para adicionar funcionalidades ao editor, como suporte para diferentes linguagens de programação, ferramentas de depuração e formatação de código.

Como profissional e professor de desenvolvimento de software, uso o VS Code como minha IDE principal e recomendo. Se você está começando sua jornada na programação ou já é um desenvolvedor experiente, o VS Code é uma ferramenta que certamente vale a pena explorar e integrar ao seu fluxo de trabalho.

Segue um tutorial simplificado de como usar o VS Code para programar em JavaScript:

1. Instale o VS Code: Acesse o site oficial do VS Code (<https://code.visualstudio.com/>) e baixe o instalador para o seu sistema operacional.
2. Abra o VS Code.
3. Clique em “File” > “Open Folder”.
4. Selecione a pasta do seu projeto e clique em “Open”.
5. Clique no ícone de “New File” ao lado do nome da pasta aberta na aba lateral esquerda.
6. Digite o nome do seu arquivo JavaScript (por exemplo, “index.js”) e pressione Enter.
7. Escreva seu código JavaScript no arquivo, por exemplo: `console.log('Olá, mundo!');`
8. Abra o terminal integrado do VS Code “Terminal” > “New Terminal”.
9. Execute o arquivo JavaScript usando o comando `bun index.js`. Este passo só funciona com o Bun instalado no seu computador, veja a próxima seção de configuração do Bun.

O idioma padrão do VS Code é o Inglês, mas pode ser alterado para Português do Brasil, basta instalar a extensão “Portuguese (Brazil)” e reiniciá-lo. Assim também como você pode customizar o tema, cores, fonte, ícones, etc.

Pesquise por extensões do VS Code para ampliar as funcionalidades e otimizar sua experiência de desenvolvimento com JavaScript e Bun. Além disso, personalize a interface do VS Code conforme suas preferências e requisitos específicos.

## 3.5. Saída de dados com `console.log()`

Para apresentar a saída de um programa em JavaScript podemos usar a instrução `console.log()`, esse comando permite que você exiba mensagens diretamente no console do navegador ou no terminal de desenvolvimento. Ela é frequentemente usada por programadores para verificar o que está acontecendo em seu código

enquanto o programa está sendo executado. **É hora de escrever seu primeiro código JavaScript e ver o poder da programação tomar forma!** Vamos escrever nosso primeiro programa para exibir a mensagem “Olá mundo!” :

```
1 console.log('Olá mundo!');
```

Neste caso, quando o programa é executado, ele mostrará a mensagem “Olá mundo!”, no console do navegador ou no terminal da ferramenta de desenvolvimento.

O `console.log` em JavaScript pode receber mais de um valor para imprimir na tela, separando cada valor por vírgula. Isso é útil quando você quer imprimir várias informações em uma única linha. Por exemplo:

```
1 console.log('Olá mundo,', 'JavaScript é', 10);
```

Ao utilizar `console.log('Olá mundo,', 'JavaScript é', 10);`, a mensagem “Olá mundo, JavaScript é 10” será exibida no console. Cada valor, separado por vírgula, será impresso na mesma linha com um espaço entre eles.

## 3.6. Conclusão

Neste capítulo, exploramos os fundamentos do JavaScript, desde suas características básicas até sua evolução ao longo do tempo. Destacamos a versatilidade e a popularidade dessa linguagem de programação, que a tornam uma escolha ideal para iniciantes e profissionais experientes.

Discutimos ferramentas e ambientes de desenvolvimento, como o VS Code e o Replit, destacando suas vantagens e usos específicos. O Replit oferece um ambiente online prático para começar a programar sem instalações, enquanto o VS Code se destaca como uma IDE robusta e altamente personalizável.

Além disso, abordamos os ambientes de execução JavaScript, como o navegador web e o Bun. Testar e depurar o código JavaScript diretamente no console do navegador é uma maneira rápida e eficiente de verificar resultados, sem necessidade de configurações adicionais. O Bun, por sua vez, surge como uma alternativa moderna ao Node.js, com instalação simples e sem a necessidade de configurações iniciais, tornando-o ideal para iniciantes.

Por fim, ressaltamos a saída de dados com a instrução `console.log()`, um recurso essencial para exibir mensagens no console do navegador ou no terminal de desenvolvimento, facilitando a depuração e verificação de resultados.

**Chegou a hora colocar a mão no código!** No próximo capítulo, você colocará a mão na massa e começará a escrever seus primeiros códigos em JavaScript. Prepare-se para uma jornada empolgante e desafiadora de aprendizado prático, onde você aprenderá na prática os fundamentos da programação com JavaScript e descobrirá todo o potencial da programação de computadores.

## 3.7. Exercícios

1. Quais as principais características do JavaScript que o tornam uma linguagem de programação popular?
2. Diferencie JavaScript de *ECMAScript*. Qual a relação entre os dois?
3. Cite e explique os principais ambientes de execução do JavaScript.
4. Pesquise as vantagens de usar o Bun como plataforma de execução do JavaScript.
5. Compare e contraste as ferramentas de desenvolvimento *Replit* e *VS Code* para JavaScript.
6. Explique a função da instrução `console.log()` em JavaScript.
7. Crie um novo projeto JavaScript no *Replit* usando o template Bun JS.
8. Escreva um código JavaScript que exiba a mensagem “Olá mundo!” no console do *Replit*.
9. Utilize o console do navegador para exibir a mensagem “Olá mundo!” usando o `console.log()`.
10. Crie um projeto JavaScript no *VS Code* que imprima a mensagem “Olá mundo!” no terminal usando `console.log()`.
11. Pesquise sobre extensões do *VS Code* para adicionar funcionalidades e aprimorar sua experiência de desenvolvimento com JavaScript e Bun.
12. Personalize as interfaces do *Replit* e do *VS Code* de acordo com suas preferências e necessidades.

## 4. Tipos de Dados e Variáveis

Neste capítulo, vamos explorar tipos de dados, variáveis e a sintaxe básica do JavaScript, além de abordar a entrada de dados com `prompt()` e a conversão de dados. Vamos entender como os computadores armazenam diferentes tipos de informações, explorar os tipos de dados disponíveis na linguagem JavaScript e aprender a declarar, atribuir valores e utilizar variáveis.

Além disso, vamos explorar como solicitar e processar informações fornecidas pelo usuário utilizando a função `prompt()`, bem como aprender a converter esses dados para os tipos adequados conforme necessário em nossos programas.

### 4.1. Tipos de Dados

Nesta seção, vamos abordar o conceito de dados relacionados à computação e seus principais tipos. Exploraremos o armazenamento e a manipulação de dados pelos computadores, além de discutir os tipos de dados específicos do JavaScript e suas particularidades.

#### 4.1.1. O que são Dados?

Em programação, dados são informações que podem ser processadas e manipuladas pelo computador. Essas informações podem representar números, textos, valores lógicos (verdadeiro ou falso), coleções de dados e muito mais. Os dados são fundamentais para a execução de operações e algoritmos em um programa de computador.

Os computadores armazenam dados em sua memória, que é organizada em locais específicos chamados de endereços de memória. Cada tipo de dado tem um tamanho e uma representação na memória do computador. Por exemplo, um número inteiro pode ser armazenado em um número fixo de bits, enquanto uma palavra de texto pode ocupar uma quantidade variável de espaço na memória, dependendo do seu comprimento.

Quando você armazena dados em um computador, eles são convertidos em bits, que são unidades básicas de informação (0 ou 1). O computador organiza esses bits em unidades maiores, como bytes, kilobytes, megabytes, gigabytes e terabytes.

### 4.1.2. Tipos de dados em JavaScript:

JavaScript possui diversos tipos de dados que podem ser utilizados para representar diferentes tipos de informações. Os principais tipos de dados são:

- **Primitivos:**

- **Number:** Números inteiros (1, 2, 3) ou decimais (3.14, 2.718).
- **String:** Sequências de caracteres entre aspas simples ou entre aspas duplas, ou seja, qualquer coisa entre aspas é uma string, como: “Aprenda Programar com JavaScript”.
- **Boolean:** Valores True ou False, usados para representar condições lógicas.
- **Undefined:** Indica que uma variável não foi inicializada.
- **Null:** Indica um valor nulo intencionalmente.
- **Symbol:** Valores únicos e imutáveis, usados para identificar propriedades de objetos.

- **Compostos:**

- **Object:** Estruturas complexas que podem armazenar coleções de dados e propriedades.
- **Array:** Listas ordenadas de valores que podem ser de qualquer tipo.
- **Function:** Blocos de código reutilizáveis que podem ser executados quando necessário.

Esses são os principais tipos de dados em JavaScript. Cada tipo de dado tem suas próprias características e é utilizado para representar diferentes tipos de informações dentro de um programa.

Utilizar o tipo de dado adequado é fundamental para a eficiência e segurança do código, em qualquer linguagem de programação. Isso ajuda o computador a

interpretar e processar os dados da maneira correta, evitando erros e problemas de desempenho.

Compreender os tipos de dados em JavaScript é fundamental, mesmo que seja uma linguagem de tipagem dinâmica. Essa compreensão permite que os desenvolvedores manipulem e interpretem os dados de forma mais precisa e confiável, evitando resultados inesperados.

Por exemplo, se você comparar duas variáveis sem saber o tipo de dado, o resultado pode ser inesperado. Da mesma forma, operações matemáticas realizadas com variáveis de tipos diferentes podem produzir resultados incorretos.

Ao compreender os tipos de dados, você pode escrever código mais eficiente, seguro, legível e fácil de manter.

## 4.2. Variáveis

No contexto da programação, uma variável é um identificador (apelido) atribuído pelo programador a um espaço de memória do computador que armazena um valor. As variáveis são fundamentais para armazenar e manipular dados em um programa, podendo conter diferentes tipos de dados, como números, strings, booleanos, entre outros.

As variáveis permitem que os programadores atribuam valores a elas durante a execução do programa e usem esses valores em várias partes do código. Elas também podem ser atualizadas e modificadas conforme necessário durante a execução do programa. As variáveis são uma parte fundamental da programação e são amplamente utilizadas em todas as linguagens de programação. Características de uma variável:

- **Nome:** Cada variável possui um nome único que a identifica no programa.
- **Tipo:** As variáveis podem armazenar diferentes tipos de dados, como números, textos, datas, etc.
- **Valor:** O valor armazenado na variável pode ser alterado durante a execução do programa.



### 4.2.1. Declaração de variáveis

Em JavaScript, você pode declarar variáveis usando as palavras-chave `var`, `let` e `const`.

- `var`: a palavra-chave mais antiga, com escopo de função.
- `let`: a palavra-chave moderna recomendada, com escopo de bloco.
- `const`: para declarar variáveis constantes (que não podem ser reatribuídas).

Veja abaixo exemplos de declarações de variáveis:

```
1 var ano = 2024;  
2  
3 let nome = 'José';  
4  
5 const PI = 3.14;
```

Vamos descrever cada item de cada linha de código do exemplo acima: Linha 1:  
`var ano = 2024;`

- `var`: É a palavra-chave utilizada para declarar uma variável. No caso, estamos declarando uma variável chamada “idade”.
- `idade`: É o nome da variável que estamos declarando.
- `=`: É o operador de atribuição, usado para atribuir um valor à variável.
- `10`: É o valor atribuído à variável “idade”. Neste caso, é um número inteiro representando a idade, que é 10.

Linha 3: `let nome = 'José';`

- `let`: É a palavra-chave utilizada para declarar uma variável.
- `nome`: É o nome da variável que estamos declarando.
- `=`: É o operador de atribuição, usado para atribuir um valor à variável.
- `'José'`: É o valor atribuído à variável “nome”. Neste caso, é uma string contendo o nome “José”, delimitada por aspas simples.

Linha 5: `const PI = 3.14;`

- `const`: É a palavra-chave utilizada para declarar uma constante. Uma vez atribuído um valor, uma constante não pode ser alterada posteriormente.
- `PI`: É o nome da constante que estamos declarando.
- `=`: É o operador de atribuição, usado para atribuir um valor à constante.
- `3.14`: É o valor atribuído à constante “PI”.

## 4.2.2. Atribuição de valores

A atribuição de valores a variáveis é uma das operações mais básicas e fundamentais da programação. Ela permite que você armazene dados na memória do computador para uso posterior em seu programa. Em JavaScript, a atribuição de valores é feita usando o operador de atribuição “=”, que atribui o valor da direita para a variável à esquerda do operador.

Por exemplo: `let nomeDaVariavel = valor;` Onde:

- `nomeDaVariavel` é o nome da variável que você deseja criar ou atualizar.
- `valor` é o valor que você deseja armazenar na variável.

A inicialização de variáveis declaradas com `let` pode ocorrer simultaneamente à sua declaração ou em momentos posteriores no código. Ao declarar uma variável, é possível atribuir imediatamente um valor a ela, como:

```
1 let cidade = 'Picos';
```

Alternativamente, é possível declarar a variável sem atribuir um valor imediato e depois atribuir um valor posteriormente no código, como:

```
1 let estado;  
2  
3 estado = 'Piauí';
```

Ambas as abordagens são válidas e úteis em diferentes situações, permitindo flexibilidade na inicialização e utilização de variáveis conforme necessário ao longo do programa.

Ao usar `const` para declarar uma variável em JavaScript, é necessário atribuir um valor imediatamente durante a declaração. Isso ocorre porque as variáveis declaradas com `const` são constantes e não podem ser reatribuídas posteriormente no código. Portanto, a inicialização de variáveis com `const` deve ocorrer simultaneamente à sua declaração:

```
1 const pais = 'Brasil';
```

Neste exemplo, a variável `pais` é declarada como uma constante e recebe imediatamente o valor `'Brasil'`. Uma vez atribuído, o valor da constante `pais` não pode ser alterado ao longo do programa.

Boas práticas para declaração de variáveis:

- Utilize nomes descritivos para suas variáveis.
- Evite usar palavras-chave reservadas como nomes de variáveis.
- Declare suas variáveis usando as palavras-chave `let` ou `const`.

### 4.2.3. Alteração de valores

Em JavaScript, as variáveis declaradas com `let` são mutáveis, o que significa que seu valor pode ser alterado após a inicialização. Essa flexibilidade é fundamental para construir programas dinâmicos e interativos. Veja o exemplo abaixo:

```
1 // Inicializando a variável nomeCompleto
2 let nomeCompleto = 'Antonio Silva';
3
4 console.log(nomeCompleto); // Saída: Antonio Silva
5
6 // Alterando o valor da variável nomeCompleto
7 nomeCompleto = 'Maria Pereira';
8
9 console.log(nomeCompleto); // Saída: Maria Pereira
```

Explicação do código:

- Na primeira linha, declaramos a variável `nomeCompleto` e a inicializamos com a string “Antonio Silva”.
- Na segunda linha, usamos `console.log` para imprimir o valor inicial da variável.
- Na terceira linha, alteramos o valor da variável `nomeCompleto` para “Maria Pereira”.
- Na quarta linha, imprimimos o novo valor da variável.

O valor de uma variável pode ser alterado quantas vezes forem necessárias.

#### 4.2.4. Usando `console.log` para exibir valores de variáveis

Podemos usar a instrução `console.log` para exibir valores de variáveis, conforme o exemplo abaixo:

```
1 let personagem = "Alice";
2 let idade = 30;
3
4 console.log("O nome da personagem é:", personagem);
5 console.log("A idade dela é:", idade);
```

Neste exemplo, nós declaramos duas variáveis: `personagem` e `idade`, e em seguida utilizamos `console.log()` para exibir uma mensagem juntamente com os valores dessas variáveis. As frases “O nome é:” e “A idade é:” são strings que aparecerão literalmente na mensagem impressa. Por outro lado, `personagem` e `idade` são variáveis que contêm valores, os quais serão exibidos no lugar das variáveis em tempo de execução. Ao executar este código, você verá no console do navegador ou no terminal de desenvolvimento as mensagens:

```
1 O nome da personagem é: Alice
2 A idade dela é: 30
```

Esse recurso é útil para verificar os valores das variáveis durante a execução do programa e para identificar possíveis erros ou comportamentos inesperados.

## 4.3. Sintaxe básica do JavaScript

Dominar a sintaxe básica de uma linguagem de programação é essencial para sua compreensão. Abordaremos alguns aspectos importantes da sintaxe do JavaScript:

### 4.3.1. Tipagem dinâmica

A tipagem dinâmica em JavaScript significa que o tipo de uma variável não precisa ser declarado explicitamente. O tipo da variável é inferido automaticamente pelo valor que é atribuído a ela. Veja o exemplo abaixo:

```
1 let livro = 'Aprenda a programar com JavaScript'
2
3 let anoDePublicacao = 2024;
```

Neste exemplo, a variável `livro` recebe uma string `Aprenda a programar com JavaScript`. Isso significa que o tipo da variável `livro` é inferido como uma string. Já a variável `livro` é inicializada com o número `2024`. Isso significa que o tipo da variável `livro` é inferido como um número inteiro.

É possível verificar o tipo de uma variável usando o operador `typeof` como demonstrado no abaixo:

```
1 let livro = 'Aprenda a programar com JavaScript';
2 console.log(typeof livro);
3
4 let anoDePublicacao = 2024;
5 console.log(typeof anoDePublicacao);
```

O operador `typeof` retorna uma string que representa o tipo da variável. Neste caso, o operador `typeof` de `livro` retorna `"string"` e de `anoDePublicacao` retorna `"number"`.

### 4.3.2. Aspas simples e duplas

Em JavaScript, aspas simples (`'`) e aspas duplas (`"`) são usadas para delimitar strings (sequências de caracteres). Veja o exemplo abaixo:

```
1 const nome = 'Steve';  
2 const sobrenome = "Jobs";  
3 console.log(nome + ' ' + sobrenome);
```

Ambas as formas são comuns e podem ser utilizadas conforme a preferência pessoal do desenvolvedor ou as convenções do código existente. Neste livro vamos usar aspas simples como convenção.

### 4.3.3. Ponto e vírgula

O ponto e vírgula (;) é um caractere especial usado em JavaScript para indicar o fim de uma instrução. No entanto, o uso do ponto e vírgula em JavaScript é opcional na maioria dos casos. Veja o exemplo de código sem ponto e vírgula:

```
1 const frase = 'Aprender JavaScript é fácil.'  
2 console.log(frase)
```

O uso do ponto e vírgula em JavaScript é opcional na maioria dos casos. No entanto, o uso do ponto e vírgula pode ser útil para evitar ambiguidade e melhorar a legibilidade do código. Neste livro vamos usar ponto e vírgula como convenção.

### 4.3.4. Comentários

Comentários em JavaScript são trechos de texto que são ignorados pelo interpretador JavaScript. Eles são usados para fornecer explicações, informações adicionais ou anotações sobre o código. Existem dois tipos de comentários em JavaScript:

- **Comentários de linha única:** Começam com // e terminam no final da linha.
- **Comentários de várias linhas:** Começam com /\* e terminam com \*/.

Veja o exemplo abaixo do uso de comentários no código:

```
1 // Este é um comentário de linha única.
2
3 /* Este é um comentário de
4 várias e
5 várias linhas. */
6
7 // Declaração e inicialização da variável 'mensagem'.
8 const mensagem = 'Venha aprender programar com o livro Aprenda a Programar\
9 com JavaScript.';
10
11 // Esta linha imprime o valor da variável 'mensagem'.
12 console.log(mensagem); /* Saída: Venha aprender programar com o livro Apre\
13 nda a Programar com JavaScript */
```

Os comentários são úteis para melhorar a legibilidade, a manutenção e a depuração do código. Use-os com moderação e de forma clara e concisa para que sejam mais eficazes.

## 4.4. Entrada de dados com prompt()

O `prompt()` é uma instrução JavaScript que facilita a solicitação de entrada de dados ao usuário, permitindo que eles digitem um valor. Essa função é valiosa para interações com o usuário, possibilitando a coleta de dados que podem ser utilizados dinamicamente em seu programa.

Ela recebe uma string entre seus parênteses, que representa a mensagem de solicitação exibida ao usuário. Em seguida, retorna a string digitada pelo usuário ou `null` se o usuário cancelar a operação. Veja o exemplo abaixo:

```
1 const nome = prompt('Digite seu nome: ');
```

No exemplo acima, quando o programa é executado, a mensagem “Digite seu nome:” é exibida na solicitação ao usuário. O valor informado pelo usuário é então armazenado na variável `nome`.

A instrução `prompt()` funciona tanto no navegador como no ambiente Bun, porém com algumas diferenças na sua forma de exibição:

- **Navegador:** O `prompt()` exibe uma caixa de diálogo modal na tela.

- **Bun:** O `prompt()` exibe a mensagem na saída padrão no terminal.

Vale destacar que o valor retornado pelo `prompt()` é sempre uma string. Se for necessária uma entrada numérica, é importante converter o valor retornado para o tipo apropriado.

No Node.js, o `prompt()` não está disponível nativamente. No entanto, é possível criar essa funcionalidade usando bibliotecas de terceiros, como o [prompt-sync](https://www.npmjs.com/package/prompt-sync)<sup>1</sup>, que funciona de forma similar ao `prompt()` do navegador e do Bun.

A função `prompt()` é frequentemente utilizada para fins de aprendizado de algoritmos e demonstração de conceitos básicos de programação. Apesar de ser uma ferramenta útil para coletar informações do usuário, **seu uso em aplicações reais não é recomendado**.

## 4.5. Conversão de tipos de dados

A conversão de tipos de dados é um conceito fundamental em programação, ela permite transformar um valor de um tipo de dado para outro. Essa conversão é essencial para realizar operações matemáticas, comparações e outras tarefas que exigem compatibilidade de tipos entre os valores envolvidos. Existem dois tipos principais de conversão de dados em JavaScript:

- **Conversão implícita:** realizada automaticamente pelo JavaScript quando necessário. Por exemplo, ao somar um número inteiro e um número de ponto flutuante, o JavaScript converte automaticamente o número inteiro para ponto flutuante.
- **Conversão explícita:** realizada pelo programador usando funções específicas. Por exemplo, a função `parseInt()` converte uma string para um número inteiro.

Exemplos de conversão implícita:

- Somar um número inteiro e um número de ponto flutuante.

---

<sup>1</sup><https://www.npmjs.com/package/prompt-sync>



```
1 let num1 = 5;
2 let num2 = 3.14;
3
4 let soma = num1 + num2; // 8.14
5
6 console.log(soma);
```

Neste exemplo, o JavaScript converte automaticamente o `num1` (inteiro) para um número de ponto flutuante para realizar a soma.

- Concatenar uma string com um número.

```
1 let str = "Olá, ";
2 let num = 5;
3
4 let mensagem = str + num; // "Olá, 5"
5
6 console.log(mensagem);
```

O JavaScript converte automaticamente o `num` para uma string para realizar a concatenação.

- Comparar um número com uma string.

```
1 let num = 10;
2 let str = "10";
3
4 let resultado = num == str; // true
5
6 console.log(resultado);
```

O JavaScript converte automaticamente a string `str` para um número para realizar a comparação.

Exemplos de conversão explícita:

- Converter uma string para um número inteiro usando `parseInt()`.

```
1  const str = "123";  
2  let num = parseInt(str); // 123  
3  
4  console.log(num);
```

- Converter uma string para um valor booleano usando `Boolean()`.

```
1  let str = "true";  
2  let bool = Boolean(str); // true  
3  
4  console.log(bool);
```

- Converter uma string para número `Number()`.

```
1  let str = "123";  
2  let num = Number(str); // 123  
3  
4  console.log(num);
```

Funções para conversão de tipos:

- `parseInt()`: Converte uma string para um número inteiro.
- `parseFloat()`: Converte uma string para um número de ponto flutuante.
- `toString()`: Converte um valor para uma string.
- `Number()`: Converte um valor para um número.
- `Boolean()`: Converte um valor para um valor booleano.

JavaScript dispõe de três formas (`Number()`, `parseInt()`, `parseFloat()`) para converter valores para números, mas cada uma tem um uso específico:

- Use `Number()` quando quiser converter **qualquer valor** para um número, mas tenha cuidado com resultados inesperados.
- Use `parseInt()` quando quiser converter uma string para um **número inteiro**, ignorando qualquer parte final não numérica.

- Use `parseFloat()` quando quiser converter uma string para um **número de ponto flutuante**, ignorando qualquer parte final não numérica.

É importante ter em mente que, ao realizar conversões de tipos de dados, é possível perder informações ou obter resultados inesperados, especialmente quando os tipos de dados são incompatíveis. Portanto, é essencial entender bem as regras de conversão de tipos da linguagem que está sendo utilizada e aplicá-las de forma adequada para evitar erros e comportamentos inesperados em seu código.

## 4.6. Conclusão

Neste capítulo, exploramos conceitos fundamentais em JavaScript, incluindo tipos de dados, variáveis e a sintaxe básica da linguagem. A tipagem dinâmica do JavaScript permite flexibilidade ao lidar com diferentes tipos de dados, enquanto o uso padronizado de aspas, ponto e vírgula e comentários são elementos essenciais da sintaxe que ajudam a estruturar e compreender o código de forma clara e organizada.

Além disso, abordamos como usar a função `prompt()` para interagir com o usuário e coletar informações, e também a importância da conversão de tipos de dados. Esta conversão é crucial para manipular e processar as informações corretamente, garantindo que estejam no formato adequado para as operações necessárias em nossos programas.

Esses conceitos são essenciais para qualquer programador que deseja dominar a linguagem e desenvolver soluções eficientes e robustas. **Parabéns por concluir este capítulo fundamental sobre JavaScript!** Prepare-se para dominar o fluxo do seu código e criar programas mais flexíveis no próximo capítulo.

## 4.7. Exercícios resolvidos

Vamos criar um programa que solicita um número ao usuário usando `prompt()`, exibe o tipo de dado da entrada fornecida usando `typeof` e, em seguida, converte essa entrada para um número usando `Number()` e por fim, o tipo de dado convertido é exibido no console usando `typeof`.

```
1 // Solicita um número ao usuário
2 const entrada = prompt("Digite um número:");
3
4 // Exibe o tipo de dado da entrada
5 console.log("O tipo de dado da entrada é:", typeof entrada);
6
7 // Converte a entrada para número usando Number()
8 const numero = Number(entrada);
9
10 // Exibe o tipo de dado convertido
11 console.log("O tipo de dado convertido é:", typeof numero);
```

## 4.8. Exercícios

1. O que é uma variável em programação? Por que são necessárias?
2. O que é tipagem dinâmica em JavaScript? Como ela difere da tipagem estática?
3. Por que é importante escolher nomes descritivos para variáveis em JavaScript? Qual é o impacto de escolher nomes inadequados?
4. Escreva um programa que utilize a instrução `console.log()` para exibir uma mensagem de boas-vindas ao usuário.
5. Qual é a diferença entre as palavras-chave `var`, `let` e `const` em JavaScript?
6. Escreva um programa que declare uma variável para armazenar um número inteiro e outra variável para armazenar um número decimal. Em seguida, imprima o tipo de dado de cada variável.
7. Utilize comentários de linha única e comentários de várias linhas em um programa simples que você escrever.
8. Escreva um programa que solicite ao usuário seu nome e exiba-o no console.
9. Escreva um programa que peça ao usuário para digitar sua idade e exiba-a no console.
10. Escreva um programa que solicite ao usuário o seu nome e sua idade e em seguida imprima essas informações juntas em uma única linha.
11. Escreva um programa que solicite ao usuário um número e converta-o para número usando `parseInt()`. Em seguida, exiba o número convertido no console.

12. Escreva um programa que peça ao usuário para digitar sua altura em metros e converta-a para número de ponto flutuante usando `parseFloat()`. Em seguida, exiba a altura convertida no console.
13. Escreva um programa que solicite ao usuário sua idade e converta-a para número usando `Number()`. Em seguida, exiba a idade no console.

# 5. Operadores e Estruturas de Controle de Fluxo

Imagine um mundo onde seu código JavaScript dança conforme sua vontade, respondendo a diferentes condições e navegando por diferentes caminhos. Esse mundo é real, e a chave para dominá-lo é o uso de **operadores e controle de fluxo**.

Neste capítulo, você embarcará em uma jornada empolgante para aprender sobre o uso de operadores e como controlar o fluxo do seu código com maestria. Descubra como:

- Operadores realizam operações de cálculos, comparações e direcionamento de fluxo.
- Estruturas condicionais como `if`, `else` e `switch` permitem tomar decisões e direcionar o fluxo do seu código como um maestro.
- Laços de repetição como `for` e `while` executam blocos de código repetidamente, automatizando tarefas complexas como um robô incansável.
- O uso de boas práticas de controle de fluxo torna seu código mais eficiente, legível e robusto como um edifício com uma estrutura sólida.

Ao dominar esses conceitos, você será capaz de criar programas mais dinâmicos e adaptáveis, capazes de lidar com uma variedade de cenários e desafios de programação.

## 5.1. Operadores

Em programação, operadores são símbolos especiais que permitem realizar operações em valores e variáveis, construindo expressões complexas e controlando o fluxo do seu código. Os operadores são fundamentais para realizar diferentes tipos de cálculos, comparações e manipulações de dados em um programa.

Os operadores em JavaScript, assim como em outras linguagens de programação, são classificados em diversos tipos, incluindo:

### Operadores aritméticos:

- `+`: Soma
- `-`: Subtração
- `*`: Multiplicação
- `/`: Divisão
- `%`: Resto da divisão
- `++`: Incremento (adiciona 1)
- `--`: Decremento (subtrai 1)

Exemplos de aplicação dos operadores aritméticos:

```
1 // Soma
2 let soma = 2 + 3;
3 console.log('soma:', soma); // Saída => soma: 5
4
5 // Subtração
6 let subtracao = 5 - 2;
7 console.log('subtracao:', subtracao); // Saída => subtracao: 3
8
9 // Multiplicação
10 let multiplicacao = 2 * 3;
11 console.log('multiplicação:', multiplicacao); // Saída => multiplicação: 6
12
13 // Divisão
14 let divisao = 6 / 2;
15 console.log('divisão:', divisao); // Saída => divisão: 3
16
17 // Resto da divisão
18 let restoDivisao = 5 % 2;
19 console.log('resto da divisão:', restoDivisao); /* Saída => resto da divis\
20 ão: 1 */
21
22 // Incremento
23 let incremento = 1;
```

```
24     incremento++;
25     console.log('incremento: ', incremento); // Saída => incremento: 2
26
27 // Decremento
28 let decremento = 2;
29     decremento--;
30     console.log('decremento:', decremento); // Saída => decremento: 1
```

Como visto nos exemplos acima, os operadores aritméticos são utilizados para realizar operações matemáticas com valores numéricos.

### Operadores de atribuição:

- `=`: Atribuição simples
- `+=`: Atribuição com adição
- `-=`: Atribuição com subtração
- `*=`: Atribuição com multiplicação
- `/=`: Atribuição com divisão
- `%=`: Atribuição com resto da divisão

### Exemplos de aplicação dos operadores de atribuição:

```
1 // Atribuição simples
2 let nome = 'Ana';
3 console.log('nome:', nome); // Saída => nome: Ana
4
5 // Atribuição com adição
6 let idade = 20;
7 idade += 2;
8 console.log('idade:', idade); // Saída => idade: 22
9
10 // Atribuição com subtração
11 let saldo = 100;
12 saldo -= 50; // saldo = 50
13 console.log('saldo:', saldo); // Saída => saldo: 50
14
15 // Atribuição com multiplicação
16 let pontuacao = 10;
```



```
17 pontuacao *= 2; // pontuacao = 20
18 console.log('pontuacao:', pontuacao); // Saída => pontuacao: 20
19
20 // Atribuição com divisão
21 let tempo = 60;
22 tempo /= 2; // tempo = 30
23 console.log('tempo:', tempo); // Saída => tempo: 30
24
25 // Atribuição com resto da divisão
26 let resultado = 10 % 3;
27 resultado %= 2; // resultado = 0
28 console.log('resultado:', resultado); // Saída => resultado: 0
```

Como demonstrado nos exemplos acima, operadores de atribuição são utilizados para atribuir valores a variáveis à esquerda.

### Operadores de comparação:

- ==: Igualdade (compara valores)
- ===: Igualdade estrita (compara valores e tipos)
- !=: Diferença (compara valores)
- !==: Diferença estrita (compara valores e tipos)
- <: Menor que
- <=: Menor ou igual que
- >: Maior que
- >=: Maior ou igual que

Exemplos de aplicação dos operadores de comparação:

```
1 // Igualdade (compara valores)
2 let igualdade = 2 == 2; // valores e tipos iguais
3 console.log('igualdade: ', igualdade); // Saída => igualdade: true
4
5 // Igualdade (compara valores)
6 igualdade = 2 == '2'; // valores iguais, mas tipos diferentes
7 console.log('igualdade: ', igualdade); // Saída => igualdade: true
8
9 // Igualdade estrita (compara valores e tipos)
10 let igualdadeEstrita = 2 === 2; // valores e tipos iguais
11 console.log('igualdadeEstrita:', igualdadeEstrita); /* Saída => igualdadeE\
12 strita: true */
13
14 // Igualdade estrita (compara valores e tipos)
15 igualdadeEstrita = 2 === '2'; // valores iguais, mas tipos diferentes
16 console.log('igualdadeEstrita:', igualdadeEstrita); /* Saída => igualdadeE\
17 strita: false */
18
19 // Diferença (compara se valores são diferentes)
20 let diferenca = 3 != 4;
21 console.log('diferenca:', diferenca); // Saída => diferenca: true
22
23 // Diferença estrita (compara valores e tipos)
24 let diferencaEstrita = 'a' !== 'a';
25 console.log('diferencaEstrita:', diferencaEstrita); /* Saída => diferencaE\
26 strita: false */
27
28 // Menor que (verifica se um valor é menor que outro)
29 let menorQue = 1 < 2;
30 console.log('menorQue:', menorQue); // Saída => menorQue: true
31
32 // Menor ou igual que (verifica se um valor é menor ou igual a outro))
33 let menorOuIgual = 2 <= 2;
34 console.log('menorOuIgual:', menorOuIgual); // Saída => menorOuIgual: true
35
36 // Maior que (verifica se um valor é maior que outro)
37 let maiorQue = 4 > 4;
38 console.log('maiorQue:', maiorQue); // Saída => maiorQue: false
39
40 // Maior ou igual que (verifica se um valor é maior ou igual a outro))
41 let maiorOuIgual = 4 >= 4;
42 console.log('maiorOuIgual:', maiorOuIgual); // Saída => maiorOuIgual: true
```

Operadores de comparação verificam a relação entre dois valores, retornando verdadeiro ou falso. Em JavaScript existem dois operadores de igualdade, que diferem na maneira como comparam valores:

- O operador de igualdade (==) compara apenas os valores de duas variáveis.
- O operador de igualdade estrita (===) compara os valores e os tipos de duas variáveis.

É recomendável usar a igualdade estrita (===) na maioria das situações, pois é mais precisa e evita resultados inesperados.

### Operadores lógicos:

- &&: E (ambas as condições precisam ser verdadeiras)
- ||: Ou (pelo menos uma das condições precisa ser verdadeira)
- !: Negação (inverte o valor da expressão)

Exemplos de aplicação dos operadores lógicos:

```
1 // E (ambas as condições precisam ser verdadeiras)
2 let condicaoE = true && true;
3 console.log('condicaoE:', condicaoE); // Saída => condicaoE: true
4
5 // E (ambas as condições precisam ser verdadeiras)
6 condicaoE = true && false;
7 console.log('condicaoE:', condicaoE); // Saída => condicaoE: false
8
9 // Ou (pelo menos uma das condições precisa ser verdadeira)
10 let condicaoOU = false || true;
11 console.log('condicaoOU:', condicaoOU); // Saída => condicaoOU: true
12
13 // Ou (pelo menos uma das condições precisa ser verdadeira)
14 condicaoOU = false || false;
15 console.log('condicaoOU:', condicaoOU); // Saída => condicaoOU: false
16
17 // Negação (inverte o valor lógico da expressão)
18 let negacao = !false; // condicao3 = true
19 console.log('negacao:', negacao); // Saída => negacao: true
```

Operadores lógicos permitem a criação de expressões complexas a partir de valores booleanos (verdadeiro ou falso), possibilitando a avaliação de condições e a tomada de decisões em programas.

### Operador de string:

- `+`: Concatenação de strings

Exemplos de aplicação do operador de string:

```
1 // Concatena (junta) duas strings literais
2 let saudacao = 'Olá ' + 'mundo!';
3 console.log('saudacao:', saudacao); // Saída => saudacao: Olá mundo!
4
5
6 let nome = 'Ana';
7 let sobrenome = 'Silva';
8 /* Concatena três strings, as variáveis nome e sobrenome e uma string com \
9 espaço entre elas */
10 let nomeCompleto = nome + ' ' + sobrenome;
11 console.log('nomeCompleto:', nomeCompleto); /* Saída => nomeCompleto: Ana \
12 Silva */
```

O operador de string, representado pelo símbolo `+` em JavaScript, é usado para concatenar duas ou mais strings, ou seja, para unir várias strings em uma única string. A ordem de precedência dos operadores em JavaScript define qual operador é executado primeiro em uma expressão. É importante conhecer essa ordem para evitar resultados inesperados. Ordem de precedência dos operadores em JavaScript:

1. Parênteses: `()`
2. Exponenciação: `**`
3. Negação unária: `!`
4. Multiplicação e divisão: `*`, `/`
5. Adição e subtração: `+`, `-`
6. Operadores de comparação: `<`, `>`, `<=`, `>=`, `==`, `===`
7. Operadores lógicos: `&&`, `||`
8. Operador de atribuição: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

Exemplo:

```
1 // A multiplicação é realizada antes da adição.
2 let resultado = 2 + 3 * 4;
3 console.log('resultado:', resultado); // Saída => resultado: 14
4
5 // Os parênteses forçam a adição a ser realizada antes da multiplicação.
6 resultado = (2 + 3) * 4;
7 console.log('resultado:', resultado); // Saída => resultado: 20
```

Boas práticas para uso de operadores:

- Use parênteses para forçar a ordem de precedência desejada.
- Evite expressões complexas com muitos operadores.
- Divida expressões complexas em partes menores e mais fáceis de entender.

A ordem de precedência dos operadores é fundamental para escrever código JavaScript correto e eficiente.

Nesta seção, exploramos os principais tipos de operadores em JavaScript, cada um com funções específicas para realizar diversas operações. No próximo tópico, abordaremos as estruturas de controle de fluxo, que permitem controlar a ordem de execução de um programa.

## 5.2. Controle de fluxo

Controle de fluxo é uma técnica fundamental na programação que permite direcionar o fluxo de execução de um programa de acordo com certas condições ou critérios. Em essência, o controle de fluxo permite que você tome decisões lógicas durante a execução do seu código, determinando quais partes serão executadas com base em variáveis, valores de retorno de funções, entradas do usuário ou outros estados do programa.

Existem duas formas principais de controle de fluxo: estruturas condicionais e estruturas de repetição também são conhecidas como laços (*loops* em inglês). As estruturas condicionais permitem que o programa tome decisões com base em condições específicas, enquanto as estruturas de repetição permitem que o programa execute um bloco de código repetidamente enquanto uma condição específica for verdadeira.

Em JavaScript, as estruturas condicionais comuns incluem o `if`, `else if` e `else`, enquanto as estruturas de repetição incluem `for`, `while` e `do-while`. Com o controle de fluxo, os programadores podem criar lógicas complexas e adaptáveis para lidar com uma variedade de cenários e problemas de programação.

As estruturas de controle são a espinha dorsal da programação, sendo cruciais para praticamente todos os programas, desde os mais simples até os mais complexos, onde são necessárias para gerenciar fluxos de execução, tomar decisões e repetir tarefas de forma eficiente.

## 5.3. Estruturas condicionais

As estruturas condicionais são fundamentais na programação, pois permitem que um programa tome decisões com base em condições específicas. Elas ajudam a controlar o fluxo de execução do código, permitindo que diferentes blocos de instruções sejam executados dependendo do resultado de uma condição.

A importância das estruturas condicionais reside na capacidade de tornar os programas mais dinâmicos e flexíveis. Com elas, é possível criar lógicas complexas e responder a diferentes situações de forma automatizada. Isso aumenta a eficiência do código e melhora a experiência do usuário, pois o programa pode se adaptar a diferentes cenários e tomar ações apropriadas com base nas condições atuais.

Existem diversos tipos de estruturas de controle condicional, sendo as mais comuns:

- `if`: (**se** em português) permite a execução de um bloco de código se uma condição específica for verdadeira.
- `else`: (**senão** em português) permite a execução de um bloco de código alternativo se a condição do `if` for falsa.
- `else if`: (**senão se** em português) permite a verificação de outras condições após um `if` inicial, possibilitando a execução de diferentes blocos de código para cada condição específica.
- `switch`: Permite a execução de um bloco de código específico com base no valor de uma variável.

### 5.3.1. Condicionais com if, else, e else if

O `if` é uma estrutura de controle de fluxo em JavaScript (e em muitas outras linguagens de programação) que permite executar um bloco de código se uma condição especificada for verdadeira. A ideia básica por trás do `if` é fazer com que o programa tome decisões com base nas condições que são avaliadas como verdadeiras ou falsas. Aqui está a sintaxe básica do `if` em JavaScript:

```
1  if (condição) {  
2    // código a ser executado se a condição for verdadeira  
3  }
```

Explicação:

- `if`: palavra-chave que indica o início da instrução condicional.
- condição: uma expressão que pode ser avaliada como verdadeira (`true`) ou falsa (`false`).
- `{}`: chaves que delimitam o bloco de código a ser executado.

Nesse caso, o código dentro das chaves só será executado se a condição for verdadeira. Se a condição for falsa, o `if` simplesmente ignora o bloco de código e continua a execução do programa. Exemplo com um `if` simples para aplicar um desconto a uma compra, dependendo do valor total da compra:

```
1  // Valor total da compra  
2  let valorCompra = parseFloat(prompt("Digite o valor da compra: R$ "));  
3  
4  let desconto = 0;  
5  // Verifica se o valor da compra é igual ou maior que 100  
6  if (valorCompra >= 100) {  
7    // 10% de desconto se o valor da compra for maior que 100  
8    desconto = 0.10;  
9  }  
10  
11 // Calculando o valor final da compra com desconto  
12 let valorFinal = valorCompra - (valorCompra * desconto);  
13  
14 // Exibindo o resultado
```

```
15 console.log(`Valor da compra: R$ ${valorCompra}`);  
16 console.log(`Desconto aplicado: ${desconto}%`);  
17 console.log(`Valor final da compra: R$ ${valorFinal}`);
```

Neste exemplo:

- Solicitamos ao usuário o valor da compra.
- O `if` é utilizado para verificar se o valor da compra é maior que 100. Se essa condição for verdadeira, atribuímos um desconto de 10% à variável `desconto`.
- Em seguida, calculamos o valor final da compra com o desconto aplicado.
- Por fim, exibimos o valor da compra original, o desconto aplicado e o valor final da compra após o desconto ser aplicado.

O `if` simples é ideal para situações onde você só precisa executar um código específico se a condição for verdadeira, como no caso de aplicar descontos em compras com valores superiores a um determinado limite.

Já quando quando temos pelo menos duas opções de execução em que precisa executar uma ou outra devemos usar o `if` e `else`.

- `if`: define a condição que precisa ser verdadeira para executar o bloco de código.
- `else`: define o bloco de código a ser executado se a condição do `if` for falsa.

A sintaxe do `if` e `else` em JavaScript é a seguinte:

```
1 if (condição) {  
2   // código a ser executado se a condição for verdadeira  
3 } else {  
4   // código a ser executado se a condição for falsa  
5 }
```

Explicação:

- `if`: palavra-chave que indica o início da instrução condicional.
- `condição`: uma expressão que pode ser avaliada como verdadeira (`true`) ou falsa (`false`).



- `{}`: chaves que delimitam o bloco de código a ser executado.
- `else`: palavra-chave opcional que indica o bloco de código a ser executado se a condição for falsa.

Com o `if` e `else` sempre um dos trechos de códigos será executado. Exemplo com `if` e `else`:

```
1  let numero = 8;
2
3  // Verifica se a divisão do número por 2 tem resto 0
4  if (numero % 2 === 0) {
5      console.log("Número par!");
6  } else {
7      console.log("Número ímpar!");
8  }
```

O código acima verifica se um número é par ou ímpar utilizando o operador de módulo (%). O `if` verifica se a divisão do número por 2 tem resto 0. Se a condição for verdadeira, executa-se o bloco de código do próprio `if`, que exibe a mensagem “Número par!”. Caso contrário, será executado o bloco de código do `else`, que exibe a mensagem “Número ímpar!”.

É possível usar várias instruções `else if` para verificar várias condições em sequência. O `else if` é uma extensão do `if` e `else` que permite adicionar condições adicionais para serem verificadas se a condição anterior não for atendida. Aqui está a sintaxe do `else if`:

```
1  if (condição1) {
2      /* bloco de código a ser executado se a condição1 for verdadeira */
3  } else if (condição2) {
4      /* bloco de código a ser executado se a condição2 for verdadeira e a c\
5  ondição1 for falsa */
6  } else {
7      /* bloco de código a ser executado se nenhuma das condições anteriores\
8  for verdadeira */
9  }
```

Com `else if`, você pode lidar com múltiplas condições de forma sequencial, fornecendo diferentes blocos de código a serem executados com base em diferentes cenários. Isso torna o código mais flexível e permite lidar com uma variedade maior de situações. Exemplo com `if`, `else if` e `else`:

```
1 let hora = 15;
2
3 if (hora < 12) {
4     console.log("Bom dia!");
5 } else if (hora < 18) {
6     console.log("Boa tarde!");
7 } else {
8     console.log("Boa noite!");
9 }
```

Neste código, o `if` verifica se a condição `hora < 12` é verdadeira; se for, imprime “Bom dia!”. Se a condição não for verdadeira, passa para o próximo `else if`, que verifica se `hora < 18`; se essa condição for verdadeira, imprime “Boa tarde!”. Se nenhuma das condições anteriores for verdadeira, executa o bloco de código do `else`, que imprime “Boa noite!”. Isso permite que a saudação seja escolhida com base no valor da variável `hora`.

### 5.3.2. Condicionais compostas

As condicionais compostas permitem verificar mais de uma condição em um único bloco de código. São estruturas de controle de fluxo que envolvem a combinação de duas ou mais condições para determinar o fluxo de execução do programa. Elas geralmente são implementadas com os operadores lógicos `&&` (AND) e `||` (OR) para avaliar múltiplas condições simultaneamente. Um exemplo comum de condicional composta é o uso do operador `&&` para verificar se duas condições são ambas verdadeiras. Por exemplo:

```
1 let idade = 25;
2 let possuiCarteira = true;
3
4 if (idade >= 18 && possuiCarteira) {
5     console.log("Pode dirigir.");
6 } else {
7     console.log("Não pode dirigir.");
8 }
```

Neste exemplo, o código verifica se a idade é maior ou igual a 18 e se a pessoa possui uma carteira de motorista (`possuiCarteira`). Somente se ambas as condições forem verdadeiras, a mensagem “Pode dirigir.” será exibida. Outro exemplo envolve o operador `||`, que verifica se pelo menos uma das condições é verdadeira. Por exemplo:

```
1 let diaSemana = "Sábado";
2
3 if (diaSemana === "Sábado" || diaSemana === "Domingo") {
4     console.log("É fim de semana!");
5 } else {
6     console.log("Não é fim de semana.");
7 }
```

Neste caso, se o `diaSemana` for igual a “Sábado” ou “Domingo”, a mensagem “É fim de semana!” será exibida.

As condicionais compostas oferecem flexibilidade e robustez ao permitir que múltiplas condições sejam avaliadas, facilitando o controle do fluxo do programa de maneira eficaz e organizada. No entanto, é importante manter a clareza e a simplicidade ao utilizar condicionais compostas, garantindo que o código seja fácil de entender e manter. Ao empregar essas estruturas, devemos buscar uma organização lógica e coesa das condições, tornando o código mais legível e eficiente.

### 5.3.3. Condicionais aninhadas

As condicionais aninhadas são estruturas que envolvem uma condicional dentro de outra. Elas são úteis quando é necessário avaliar várias condições em diferentes níveis de prioridade. Por exemplo:

```
1 const numero = 12;
2
3 if (numero % 2 === 0) {
4     if (numero % 3 === 0) {
5         console.log("O número é par e divisível por 3!");
6     } else {
7         console.log("O número é par, mas não é divisível por 3!");
8     }
9 } else {
10     console.log("O número é ímpar!");
11 }
```

Nesse exemplo, há uma condicional aninhada dentro de outra. Primeiro, verificamos se o número é par. Se for, avaliamos se ele é divisível por 3. Dependendo dessas condições, diferentes mensagens são exibidas. Condicionais aninhadas fornecem uma maneira eficaz de lidar com casos complexos de decisão,

permitindo que múltiplas verificações sejam realizadas de forma organizada e estruturada, garantindo uma lógica clara e compreensível no código. No entanto, é importante usá-las com cuidado, pois o aninhamento excessivo pode tornar o código difícil de entender e manter.

### 5.3.4. Condicionais com switch

A estrutura `switch` é útil quando precisamos avaliar uma expressão em relação a múltiplos valores possíveis e executar diferentes blocos de código com base nesses valores. Ele oferece uma alternativa mais limpa e eficiente do que uma série de instruções `if-else`, especialmente em situações com muitas condições. Sintaxe da estrutura do `switch`:

```
1  switch (expressao) {  
2      case valor1:  
3          // código a ser executado se a expressao for igual a valor1  
4          break;  
5      case valor2:  
6          // código a ser executado se a expressao for igual a valor2  
7          break;  
8      // mais casos podem ser adicionados conforme necessário  
9      default:  
10         // código a ser executado se nenhum dos casos anteriores for verdadeiro  
11 }
```

Explicação da sintaxe:

- `switch`: palavra-chave que inicia a instrução.
- expressão: valor que será comparado com os casos.
- `case`: palavra-chave que identifica cada caso.
- valor: valor a ser comparado com a expressão.
- `break`: palavra-chave que faz com que o `switch` saia do bloco após a execução do código do caso.
- `default`: bloco opcional que é executado se a expressão não corresponder a nenhum caso.

Segue um exemplo do uso da estrutura `switch` para verificar o valor da variável `diaDaSemana` e imprimir uma mensagem diferente para cada dia da semana.

```
1  let diaDaSemana = prompt("Digite o dia da semana: ");
2
3  switch (diaDaSemana) {
4      case "segunda-feira":
5          console.log("Hoje é segunda-feira!");
6          break;
7      case "terça-feira":
8          console.log("Hoje é terça-feira!");
9          break;
10     case "quarta-feira":
11         console.log("Hoje é quarta-feira!");
12         break;
13     case "quinta-feira":
14         console.log("Hoje é quinta-feira!");
15         break;
16     case "sexta-feira":
17         console.log("Hoje é sexta-feira!");
18         break;
19     case "sábado":
20         console.log("Hoje é sábado!");
21         break;
22     case "domingo":
23         console.log("Hoje é domingo!");
24         break;
25     default:
26         console.log("Dia inválido!");
27 }
```

Explicação do código acima:

- O programa solicita ao usuário que insira o dia da semana usando o `prompt()` e armazena essa entrada na variável `diaDaSemana`.
- Em seguida, o código utiliza a estrutura `switch` para avaliar o valor de `diaDaSemana`.
- Dependendo do valor de `diaDaSemana`, o código executa um bloco de código específico correspondente ao caso. Por exemplo: se `diaDaSemana` for “segunda-feira”, ele exibirá “Hoje é segunda-feira!” no console. O mesmo se aplica para os outros dias da semana até “domingo”.
- Se o valor de `diaDaSemana` não corresponder a nenhum dos casos definidos (por exemplo, se o usuário inserir um valor inválido), o código executará o bloco

de código no caso `default`, que exibe “Dia inválido!” no console.

A estrutura `switch` é uma ferramenta eficiente para tomar decisões com base em uma única expressão que pode corresponder a vários valores predefinidos. Nesses casos ela oferece uma forma mais concisa e legível de lidar com a execução de múltiplas condições em comparação com instruções `if` aninhadas.

## 5.4. Laços de repetição

Laços de repetição são utilizados para executar um bloco de código várias vezes, com base em uma condição específica. Elas são fundamentais quando se deseja executar tarefas que exigem repetição de instruções. A importância das estruturas de repetição reside na sua capacidade de reduzir a redundância no código, aumentar a eficiência e melhorar a produtividade do desenvolvedor, permitindo a execução de tarefas repetitivas de forma rápida e eficaz.

Imagine que você tenha que desenvolver um programa para imprimir os números de 1 a 10 em linhas separadas no console. Uma forma de fazer isso é escrevendo a instrução `console.log()` para cada número de 1 a 10. Aqui está um exemplo:

```
1 console.log(1);
2 console.log(2);
3 console.log(3);
4 console.log(4);
5 console.log(5);
6 console.log(6);
7 console.log(7);
8 console.log(8);
9 console.log(9);
10 console.log(10);
```

Isso imprimirá os números de 1 a 10 em linhas separadas no console. No entanto, esse método não é escalável e pode se tornar impraticável para uma grande quantidade de números. O emprego de laços de repetição se mostra uma solução mais eficaz nessas situações.

Os laços de repetição mais comuns na programação são o “for” e o “while”. A seguir, vamos explorar ambas as estruturas.

### 5.4.1. Laço for

O laço (loop) for é uma estrutura de repetição que permite executar um bloco de código um número específico de vezes. Ele possui três partes principais que fica entre parênteses e separadas por vírgulas: a inicialização, a condição de continuação e a expressão de incremento. Sintaxe básica:

```
1 for ([inicialização]; [condição]; [incremento]) {  
2   // bloco de código a ser executado  
3 }
```

Explicação da estrutura completa do laço for:

- for: Palavra reservada que indica o início de um laço de repetição.
- Inicialização: Esta parte é executada apenas uma vez, no início do laço. É onde você define a variável que será usada para controlar a quantidade de repetições.
- Condição: Esta parte é verificada antes de cada iteração do laço. Se a condição for verdadeira, o bloco de código é executado. Se a condição for falsa, o laço termina.
- Incremento: Esta parte é executada após cada repetição do laço. É onde você modifica a variável usada na condição para controlar quantidade de repetições do laço.
- Corpo do laço: É o bloco de código que será executado a cada iteração do laço, é delimitado por chaves {} e pode conter uma ou mais instruções.

Vamos refazer o exemplo de imprimir os números de 1 a 10 usando o laço for:

```
1 for (let i = 1; i <= 10; i++) {  
2   console.log(i);  
3 }
```

Neste exemplo, o laço for é usado para exibir os números de 1 a 10 no console. A inicialização `let i = 1` define a variável de controle `i` e a inicializa com 1. A condição `i <= 10` verifica se `i` é menor ou igual a 10. Enquanto essa condição for verdadeira, o bloco de código dentro do for será executado. Após cada iteração, a expressão de incremento `i++` aumenta o valor de `i` em 1.

Além de executar um bloco de código várias vezes sem precisar reescrever o código para cada iteração, o laço `for` também é bastante utilizado para ler, analisar e modificar dados em arrays, listas, strings ou outros objetos iteráveis, como veremos no Capítulo 7.

O laço `for` é uma estrutura de controle muito útil quando você precisa repetir um bloco de código um número específico de vezes ou quando deseja percorrer uma sequência de elementos, como os elementos em um array. Aqui estão alguns cenários comuns em que o laço `for` é útil:

- **Iteração sobre elementos de um array:** Quando você precisa percorrer todos os elementos de um array e realizar uma operação em cada um deles (Capítulo 7).
- **Contagem conhecida de iterações:** Quando você sabe exatamente quantas vezes deseja repetir uma ação, por exemplo, para executar uma operação um número específico de vezes.
- **Iteração sobre uma sequência numérica:** Quando você precisa iterar sobre uma sequência de números em uma ordem específica, como de 0 a 10 ou de 1 a 100.

O laço `for` é uma ferramenta poderosa em programação, oferecendo uma maneira eficiente de executar operações repetidas por um número específico de vezes. Dominar o uso do laço `for` é fundamental para desenvolver habilidades fundamentais de programação.

#### Dicas para escrever laços de repetição:

- Escolha nomes de variáveis descritivos para facilitar a compreensão do código.
- Use indentação para organizar o código e facilitar a leitura.
- Teste o código cuidadosamente para garantir que ele funcione como esperado.
- Caso necessário, use comentários para explicar o que o código faz.

### 5.4.2. Laço `while`

O laço (*loop*) `while` é uma estrutura de repetição que permite executar um bloco de código enquanto uma condição for verdadeira. Sua sintaxe é a seguinte:



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

A condição é uma expressão avaliada antes de cada iteração do laço. Se a condição for verdadeira, o bloco de código dentro das chaves será executado. Após a execução do bloco de código, a condição será avaliada novamente. Se ainda for verdadeira, o bloco será executado novamente, e assim por diante, até que a condição se torne falsa. Quando a condição se torna falsa, a execução do loop é interrompida e o controle passa para a próxima instrução após o bloco `while`.

É importante garantir que a condição seja eventualmente falsa para evitar laços infinitos.

O `while` é útil quando o número de iterações não é conhecido antecipadamente e depende da lógica ou da entrada do usuário. Um exemplo prático de uso do `while` em JavaScript seria para solicitar ao usuário que insira um número positivo e, enquanto o número inserido for menor ou igual a zero, continuar pedindo a entrada. Aqui está o código de exemplo:

```
1 let numero = 0;  
2  
3 while (numero <= 0) {  
4   numero = parseInt(prompt("Digite um número positivo: "));  
5 }  
6  
7 console.log("O número positivo digitado é: " + numero);
```

Neste exemplo, o programa solicita ao usuário que insira um número positivo, o qual não pode ser negativo nem neutro (ou seja, deve ser maior que zero). Enquanto o número inserido for menor ou igual a zero, o programa continuará solicitando a entrada. Uma vez que o usuário fornece um número positivo válido, o laço `while` é encerrado e o programa exibe uma mensagem com o número positivo inserido.

O laço `while` é mais adequado quando não sabemos exatamente quantas vezes precisamos repetir um bloco de código, mas queremos continuar a repetição enquanto uma condição específica for verdadeira. Ele é útil em situações onde a quantidade de iterações pode variar e é determinada dinamicamente durante a execução do programa. Por exemplo, validar entrada do usuário até que uma condição específica seja atendida.

A principal diferença entre `for` e `while` reside na sua estrutura e no momento em que a condição é avaliada. Dessa forma:

- Utilize o `for` quando o número de iterações é conhecido ou pode ser determinado previamente.
- Prefira o `while` quando o número de iterações não é fixo, especialmente se a condição de parada depende de uma variável que pode ser alterada dentro do corpo do laço.

JavaScript oferece diversas estruturas de repetição além do `for` e do `while` abordados neste livro. O capítulo 7 irá explorar opções mais modernas e eficientes como `forEach`, `for...of` e `for...in`. Embora o `do...while` seja uma estrutura de repetição válida em JavaScript, optou-se por não abordá-la neste livro devido ao seu uso menos frequente, sendo facilmente substituível por um `while`.

O domínio do uso do `for` e do `while` é fundamental para criar programas eficientes e controlar fluxos de repetição em JavaScript. Cada estrutura oferece vantagens específicas, permitindo que os desenvolvedores escolham a mais adequada para suas necessidades de implementação. Use a flexibilidade de ambos para construir código robusto, eficiente e legível, abrindo caminho para soluções inovadoras e automatizadas.

### 5.4.3. Laços aninhados

Os laços aninhados, que consistem em incorporar um laço dentro de outro, são úteis para lidar com estruturas de dados bidimensionais, como matrizes. Podemos usar um laço externo para percorrer as linhas e um laço interno para percorrer as colunas. Aqui está um exemplo de laços aninhados utilizando o `for`:

```
1  for (let i = 0; i < 4; i++) {  
2      let linha = '';  
3      for (let j = 0; j < 3; j++) {  
4          linha += 'X '  
5      }  
6      console.log(linha);  
7  }
```

Este código utiliza dois laços `for` aninhados para imprimir uma matriz 4x3 composta por “X”, em que:

- O primeiro `for` cria as linhas da matriz (executa 4 vezes para criar 4 linhas).
- O segundo `for` cria as colunas em cada linha (executa 3 vezes para criar 3 colunas em cada linha).

Os laços aninhados, `for` ou `while`, oferecem uma maneira poderosa de iterar sobre elementos em duas ou mais dimensões. No entanto, é importante usá-los com cautela, pois o aninhamento excessivo pode tornar o código difícil de entender e dar manutenção.

## 5.5. Conclusão

O domínio dos operadores e das estruturas de controle de fluxo é fundamental para construir programas robustos e flexíveis. Ao compreender os diferentes tipos de operadores e como utilizá-los, você pode realizar cálculos complexos, tomar decisões e controlar o fluxo de execução do seu código.

As estruturas de controle de fluxo permitem que seus programas respondam a diferentes condições e executem ações específicas em cada caso. Através do uso de estruturas como `if`, `else`, `for` e `while`, você pode criar programas dinâmicos e adaptáveis a diversas situações.

Ao combinar o conhecimento de operadores, estruturas de controle e as boas práticas de programação, você estará pronto para construir código mais eficiente, legível e profissional.

No próximo capítulo, mergulharemos ainda mais fundo no universo da programação JavaScript, explorando o poder das funções. Descubra como as funções podem modularizar o seu código, promovendo a reutilização e a

organização, e como o escopo influencia a visibilidade e a acessibilidade das variáveis no seu programa. Prepare-se para expandir ainda mais seus horizontes e aprimorar suas habilidades de programação.

## 5.6. Exercícios resolvidos

1. Vamos criar um programa que lê 3 números, calcula a média e exibe o resultado.

```
1 // Ler o primeiro número usando o prompt()
2 const entrada1 = prompt('Digite o primeiro número: ');
3 // Ler o segundo número usando o prompt()
4 const entrada2 = prompt('Digite o segundo número: ');
5 // Ler o terceiro número usando o prompt()
6 const entrada3 = prompt('Digite o terceiro número: ');
7
8 /* Converte entrada1 para número e armazena na variável num1 */
9 const num1 = Number(entrada1);
10 /* Converte entrada2 para número e armazena na variável num2 */
11 const num2 = Number(entrada2);
12 /* Converte entrada3 para número e armazena na variável num3 */
13 const num3 = Number(entrada3);
14
15 // Calcula a média e armazena na variável média
16 const media = (num1 + num2 + num3) / 3;
17
18 // Exibe o resultado usando o console.log
19 console.log('A média dos três números é:', media);
```

2. Agora vamos desenvolver um programa que solicita a entrada de três números, calcula a média e exibe “Aprovado” para médias iguais ou superiores a 7, “Recuperação” para médias entre 4 e 6,9 (inclusive) e “Reprovado” para médias abaixo de 4.

```
1 // Ler o primeiro número usando o prompt()
2 const entrada1 = prompt('Digite o primeiro número: ');
3 // Ler o segundo número usando o prompt()
4 const entrada2 = prompt('Digite o segundo número: ');
5 // Ler o terceiro número usando o prompt()
6 const entrada3 = prompt('Digite o terceiro número: ');
7
8 /* Converte entrada1 para número e armazena na variável num1 */
9 const num1 = Number(entrada1);
10 /* Converte entrada2 para número e armazena na variável num2 */
11 const num2 = Number(entrada2);
12 /* Converte entrada3 para número e armazena na variável num3 */
13 const num3 = Number(entrada3);
14
15 // Calcula a média e armazena na variável média
16 const media = (num1 + num2 + num3) / 3;
17
18 // Validar a média e exibir a mensagem correspondente
19 if (media >= 7) {
20     console.log('Aprovado');
21 } else if (media >= 4) {
22     console.log('Recuperação');
23 } else {
24     console.log('Reprovado');
25 }
```

3. O programa a seguir solicita um número positivo ao usuário, converte a entrada para um número inteiro usando `parseInt`, e então utiliza um laço `for` para exibir os números de 0 até o número informado. Cada número é exibido no console.

```
1 // Solicita um número ao usuário
2 const numero = parseInt(prompt('Digite um número positivo :'));
3
4 // Utiliza um laço for para exibir os números de 0 até o número informado
5 for (let i = 0; i <= numero; i++) {
6     console.log(i);
7 }
```

4. Esse programa solicita um número positivo ao usuário, converte a entrada para um número inteiro usando `parseInt`, e então utiliza um laço `while` para

exibir os números de 0 até o número informado. O contador é incrementado a cada iteração até que atinja o número informado. Cada número é exibido no console.

```
1 // Solicita um número ao usuário
2 const numero = parseInt(prompt('Digite um número positivo: '));
3
4 // Inicializa o contador
5 let contador = 0;
6
7 // Utiliza um laço while para exibir os números de 0 até o número informado
8 while (contador <= numero) {
9     console.log(contador);
10    contador++;
11 }
```

## 5.7. Exercícios

1. Implemente um programa que determine se um número fornecido pelo usuário é par ou ímpar.
2. Escreva um programa que peça ao usuário para inserir três números e, em seguida, imprima o maior deles.
3. Crie um programa que recebe três notas, calcula a média e informe se o aluno foi aprovado (média maior ou igual a 7) ou reprovado (média menor que 7).
4. Desenvolva um programa que receba o ano de nascimento de uma pessoa e informe se ela já é maior de idade ou não.
5. Faça um programa que converte uma temperatura de Celsius para Fahrenheit ou vice-versa, dependendo da escolha do usuário.
6. Escreva um programa que solicite dois números ao usuário e verifique se pelo menos um deles é múltiplo do outro. Se pelo menos um for múltiplo, exiba a mensagem “Pelo menos um dos números é múltiplo do outro”. Caso contrário, exiba a mensagem “Nenhum dos números é múltiplo do outro”.
7. Escreva um programa que peça ao usuário o valor de três lados de um triângulo e classifique-o como triângulo equilátero, isósceles, escaleno ou inválido.

8. Crie um programa que simula um restaurante. O usuário escolhe um prato do menu (opções: pizza, hambúrguer, salada, macarrão) usando a instrução `switch`. Para cada prato escolhido, exiba o preço e a descrição do prato.
9. Escreva um programa que peça ao usuário sua altura e peso, calcule o IMC (Índice de Massa Corporal) e classifique de acordo com a tabela da OMS (abaixo do peso, peso normal, sobrepeso, obesidade). Exiba o valor do IMC e a classificação usando a instrução `switch`.
10. Escreva um programa que solicite ao usuário um número positivo e exiba todos os números **pares** de 0 até o número informado (use o laço de repetição `while`).
11. Escreva um programa que solicite ao usuário um número positivo e exiba todos os números **ímpares** de 0 até o número informado (use o laço de repetição `for`).
12. Crie um programa que calcule o fatorial de um número fornecido pelo usuário.
13. Implemente um jogo de adivinhação onde o computador gera um número aleatório entre 1 e 100 e o usuário tem que adivinhar qual é em até 10 tentativas. Verifique se cada palpite do usuário está correto, menor ou maior que o número secreto. Exiba mensagens informando o resultado de cada tentativa e forneça pistas (menor/maior) para ajudar o usuário. Para gerar um número aleatório entre 0 e 100 em JavaScript podemos usar `Math.floor(Math.random() * 101)`, onde `Math.random()` retorna um valor entre 0 (inclusive) e 1 (exclusivo), que é multiplicado por 101 para incluir o 100, e `Math.floor()` arredonda o resultado para o número inteiro mais próximo.

## 6. Funções e Escopo

Neste capítulo, exploraremos em detalhes o conceito de funções, desde sua declaração e chamada até o uso de parâmetros, retorno e escopos. Além disso, discutiremos as vantagens do uso de funções e os diferentes tipos de escopos em JavaScript, destacando como esses conceitos fundamentais contribuem para a estruturação e legibilidade do código. Ao final deste capítulo, você terá uma compreensão sólida das funções e sua importância na construção de aplicativos robustos e escaláveis.

### 6.1. Funções

Uma função é um bloco de código delimitado por um início e um fim, identificado por um nome único. Esse nome serve como uma referência que permite executar a função em qualquer parte do programa onde ela seja necessária.

Uma função executa uma tarefa específica quando é chamada. Ela pode receber dados de entrada, realizar algum processamento com esses dados e pode retornar um resultado.

As funções são cruciais na programação, pois agrupam conjuntos de instruções que podem ser chamados repetidamente em diferentes partes do programa, permitindo modularidade e reutilização de código.

Além disso, dividir o código em funções com nomes descritivos facilita a compreensão e a manutenção do programa.

Essas características tornam as funções essenciais no desenvolvimento de software, contribuindo para o desenvolvimento de códigos mais organizados, legíveis e robustos.

A seguir examinaremos as funções predefinidas disponíveis em JavaScript, detalharemos a declaração e execução de funções, e discutiremos os diferentes tipos de funções, fornecendo uma compreensão abrangente do seu uso e importância na programação.



## 6.2. Funções predefinidas

Toda linguagem de programação, incluindo JavaScript, disponibiliza um conjunto de funções predefinidas que podem ser usadas sem a necessidade de serem criadas pelo programador. Essas funções facilitam o desenvolvimento, pois fornecem funcionalidades básicas e comuns, como:

- **Entrada e saída de dados:** funções como `console.log()` e `prompt()` permitem interagir com o usuário, imprimindo mensagens e coletando informações, respectivamente.
- **Manipulação de dados:** funções como `parseInt()` e `parseFloat()` convertem strings em números inteiros e decimais, respectivamente.

Essas funções predefinidas facilitam o desenvolvimento de aplicativos, pois fornecem funcionalidades comuns prontas para uso, economizando tempo e esforço do desenvolvedor. Além dessas, existem muitas outras funções pré-definidas em JavaScript, cada uma com sua finalidade específica, permitindo aos programadores criar aplicativos complexos e poderosos com mais facilidade e eficiência.

Segue um exemplo de uso das funções predefinidas `prompt()`, `parseInt()` e `console.log()`:

```
1 // Solicita um número ao usuário
2 const entrada = prompt('Digite um número: ');
3
4 // Converte a entrada para um número inteiro usando parseInt()
5 const numero = parseInt(entrada);
6
7 // Exibe o número no console
8 console.log('O número digitado é:', numero);
```

Neste código:

- A função `prompt()` é utilizada para solicitar um número ao usuário.
- O valor digitado pelo usuário é armazenado na variável `entrada`.
- Em seguida, a função `parseInt()` é utilizada para converter a entrada em um número inteiro.

- O número convertido é armazenado na variável `numero`.
- Por fim, a função `console.log()` é utilizada para exibir o número no console.

Este exemplo ilustra o uso das funções predefinidas para interagir com o usuário, converter tipos de dados e exibir informações no console.

## 6.3. Declaração de funções

A declaração de funções em JavaScript pode ser feita de algumas maneiras. Uma das formas mais comuns é utilizando a palavra-chave `function`, seguida do nome da função, parênteses e chaves que contêm as instruções. Exemplo de uma função simples sem parâmetros e sem retorno:

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

### Explicação:

- `function`: Palavra-chave que indica o início da declaração de uma função.
- `saudacao`: Nome da função, que deve ser descritivo e fácil de lembrar.
- `()`: Parênteses que podem conter parâmetros (opcional).
- `{ }`: Chaves que delimitam o corpo da função, contendo as instruções a serem executadas.

### 6.3.1. Funções com Parâmetros

As funções em JavaScript podem receber parâmetros (dados de entrada), que são valores fornecidos quando a função é chamada. Esses parâmetros são como variáveis, armazenando os valores passados para uso interno da função. Dentro da função, esses valores podem ser utilizados para realizar operações ou cálculos específicos. Aqui está um exemplo de como uma função com parâmetros pode ser declarada:

```
1 // Declaração da função nomeada "saudacao"
2 function quadrado(numero) {
3     console.log(numero**2);
4 }
```

Explicação da declaração da função `saudacao`:

- `function`: Palavra-chave que indica o início da declaração de uma função.
- `quadrado`: Nome da função, que deve ser descritivo e fácil de lembrar.
- `(numero)`: Parênteses que contêm os parâmetros da função. Neste caso, a função possui um único parâmetro chamado `numero`.
- `{ }`: Chaves que delimitam o corpo da função, contendo as instruções a serem executadas.
- `console.log(numero**2);`: Instrução que exibe o quadrado do número recebido como parâmetro.

### 6.3.2. Funções com retorno

Uma função com retorno é uma função que, ao ser executada, produz e retorna um valor específico para a parte do código que a invocou. Esse retorno é feito por meio da palavra-chave “`return`”. Toda função que possui o “`return`”, ela retorna um valor após sua execução. Exemplo de declaração de função com retorno:

```
1 // declaração da função nomeada de "soma"
2 function soma(a, b) {
3     return a + b;
4 }
```

Explicação:

- `function soma(a, b)`: Esta linha declara uma função chamada `soma`. Ela aceita dois parâmetros: `a` e `b`. Esses parâmetros representam os números que serão somados.
- `return a + b;`: Esta linha é o cerne da função. Ela soma os valores dos parâmetros `a` e `b` usando o operador `+`. O resultado da soma é então retornado pela função usando a palavra-chave `return`.

Essa função soma é uma calculadora simples de adição de dois números. Ela recebe dois números como entrada (por meio dos parâmetros a e b) e retorna a soma desses números.

### 6.3.3. Funções anônimas tradicionais e *arrow functions*

Uma função anônima é uma função que não possui um nome associado a ela. Ela pode ser declarada usando uma expressão de função ou uma arrow function sem atribuir um nome a ela. Aqui está um exemplo de declaração de uma expressão de função anônima:

```
1  const soma = function(a, b) {  
2      return a + b;  
3  };
```

Também é possível declarar funções anônimas no formato usando *arrow function* (ou funções de seta), que é uma forma concisa e moderna de escrever funções em JavaScript (introduzida no ES6 - ECMAScript 2015). Elas oferecem uma sintaxe mais curta e limpa em comparação com as funções tradicionais. Veja abaixo como fica a função criada acima usando *arrow function*:

```
1  const soma = (a, b) => {  
2      return a + b;  
3  };
```

Toda *arrow function* é anônima por natureza. Isso significa que elas não possuem um nome associado como as funções tradicionais. Elas geralmente são atribuídas a uma variável ou passadas como argumento para outras funções sem um nome identificador. Portanto, enquanto as funções tradicionais podem ser tanto nomeadas quanto anônimas, as *arrow functions* são sempre anônimas por definição.

As funções anônimas podem ser atribuídas a variáveis ou passadas como argumentos para outras funções, mas não têm um nome próprio para serem chamadas diretamente. Embora as funções anônimas não tenham um nome próprio, elas podem ser chamadas pelo nome da variável à qual foram atribuídas. Por exemplo:

```
1  const soma = function(a, b) {  
2      return a + b;  
3  };  
4  
5  console.log(soma(2, 3)); // Saída: 5
```

No exemplo acima, a função anônima é atribuída à variável `soma`, e podemos chamá-la usando o nome da variável seguido pelos parênteses para passar os argumentos.

### 6.3.4. Sintaxe de *arrow functions*

*Arrow functions* é uma forma mais concisa e moderna de escrever funções em JavaScript. A sintaxe de *arrow functions* é definida usando uma seta (`=>`) entre a lista de parâmetros (se houver) e o bloco de código da função. Veja um exemplo de sua sintaxe básica:

```
1  (parametros) => expressao
```

Neste exemplo:

- `(parametros)`: Parênteses opcionais que podem conter os parâmetros da função. Se houver apenas um parâmetro, os parênteses podem ser omitidos.
- `=>`: Símbolo de seta que separa os parâmetros da expressão.
- `expressao`: A expressão (ou corpo da função) que define o que a função irá retornar.

Abaixo estão algumas formas válidas de declaração de *arrow functions*:

```
1 // Sem parâmetros
2 const funcaoSemParametros = () => {
3     // corpo da função
4 };
5
6 // Com um parâmetro
7 const funcaoComUmParametro = (parametro) => {
8     // corpo da função
9 };
10
11 // Com múltiplos parâmetros
12 const funcaoComMultiplosParametros = (parametro1, parametro2) => {
13     // corpo da função
14 };
15
16 // Com parâmetros e retorno implícito, função recebe os parâmetros a e b e\
17 retorna a soma deles
18 const funcaoComRetornoImplicitoEParenteses = (a, b) => a+b;
19
20 // Com parâmetros e retorno explícito, função recebe os parâmetros a e b e\
21 retorna a soma deles
22 const funcaoComRetornoExplicito = (a, b) => {
23     return a + b;
24 }
25
26 // Com um parâmetro e retorno implícito, função recebe o parâmetro n e ret\
27 orna n elevado ao quadrado
28 const funcaoComRetornoImplicitoSemParenteses = n => n**2;
```

A seta => substitui a necessidade da palavra-chave `function`, e o corpo da função é definido após a seta. Se a função possuir apenas um parâmetro, os parênteses podem ser omitidos. Além disso, se houver apenas uma instrução de retorno, as chaves também podem ser omitidas, proporcionando uma sintaxe mais concisa.

Use *arrow functions* para funções simples, tornando o código mais enxuto e legível.

## 6.4. Execução de funções

A execução de funções em JavaScript é o processo de invocar uma função para executar o código que ela contém. Quando uma função é criada, ela não é executada automaticamente; ela precisa ser chamada explicitamente para que seu código seja executado. É como apertar o botão “play” para dar vida a o bloco de instruções da função. Como fazer uma chamada de função?

- **Nome da função:** Identifique a função que você deseja executar.
- **Parênteses:** Inclua os parênteses após o nome da função.
- **Argumentos (opcional):** Dentro dos parênteses, liste os valores que você deseja passar para a função, separados por vírgulas.
- **Ponto e vírgula (opcional):** Finalize a chamada da função com um ponto e vírgula.

Para chamar uma função em JavaScript, você simplesmente usa o nome da função seguido por parênteses “()”. Por exemplo:

```
1 // Declaração da função saudacao
2 function saudacao() {
3     console.log("Olá, mundo!");
4 }
5
6 /* Chamada da função saudacao, que irá exibir na tela a mensagem "Olá, mun\
7 do!" */
8 saudacao();
```

Muitas vezes, uma função precisa de dados para realizar seu trabalho. Esses dados são chamados de argumentos e são passados para a função entre os parênteses durante a chamada da função. Por exemplo:

```
1 // Declaração da função soma
2 const soma = function(a, b) {
3     // Retorna a soma de a + b
4     return a + b;
5 };
6
7 /* Chamada da função "soma" com argumentos 2 e 3 que serão atribuídos aos \
8 parâmetros a e b respectivamente da função soma.
9 A função soma irá retornar o valor da soma (5), que será armazenado na vari\
10 ável resultado.
11 */
12 const resultado = soma(2, 3);
```

No contexto da programação é comum utilizarmos os termos parâmetro e argumento como sinônimos, porém conceitualmente eles são diferentes. Entenda a diferença entre parâmetro, argumento e retorno em funções:

- **Parâmetro:** São variáveis declaradas dentro dos parênteses da função, que espera receber um valor quando a função for chamada.
- **Argumento:** São os valores reais, passados entre parênteses, na chamada da função.

O uso desses termos como sinônimos é resultado da prática comum de se referir aos valores passados para uma função durante sua chamada. Essa flexibilidade é aceita na comunidade de programação. No entanto, é importante entender as diferenças conceituais entre eles.

Pontos importantes sobre a execução de funções:

- Ao chamar uma função, é importante fornecer os valores (argumentos) corretos na ordem correta.
- O número de argumentos passados para a função deve ser igual ao número de parâmetros definidos na função.
- Se você não fornecer um valor válido para um parâmetro obrigatório, um erro será gerado.

A chamada de funções é o processo de executar o código contido dentro delas, permitindo realizar tarefas específicas de forma organizada, eficiente e reutilizável.



As funções podem ser chamadas várias vezes em diferentes partes do programa, permitindo aproveitar o mesmo conjunto de instruções para realizar tarefas similares em diversos contextos, o que contribui para a modularidade e a manutenibilidade do código.

## 6.5. Escopo

Em programação, o escopo define a área do código onde uma variável é visível e pode ser utilizada. Imagine que cada variável tem sua própria área de atuação dentro do código.

Se uma variável está dentro de uma função, por exemplo, ela só pode ser usada dentro dessa função - isso é o escopo **local**. Já se uma variável é declarada fora de qualquer função, ela pode ser usada em qualquer lugar do programa - isso é o escopo **global**. Em resumo, o escopo define quem pode “ver” e “usar” cada variável.

Agora, vamos dar uma olhada em algumas características dos escopos no JavaScript: **Escopo Global**:

- Em JavaScript, variáveis declaradas fora de qualquer função são globais.
- É o escopo mais amplo em um programa.
- As variáveis declaradas no escopo global são visíveis em todo o programa.
- Essas variáveis podem ser acessadas de qualquer lugar no código, ou seja, em qualquer função ou bloco de código.

### Escopo Local:

- Em JavaScript, variáveis declaradas dentro de uma função ou um bloco de código (como `if`, `else`, `for`, `while`) são locais.
- É o escopo de uma função ou bloco de código.
- As variáveis declaradas no escopo local são visíveis apenas dentro da função ou bloco de código onde foram declaradas.
- Variáveis locais são criadas quando a função é chamada e são destruídas quando a função é concluída.

Exemplo que ilustra o uso de variáveis globais, locais de função e locais de bloco de código em JavaScript:

```
1 // Variável global
2 let globalVariable = "Global";
3
4 function minhaFuncao() {
5     // Variável local de função
6     let localFunctionVariable = "Local de Função";
7     console.log("Dentro da função:");
8     console.log("Variável global:", globalVariable); // Acesso permitido à
9     variável global dentro da função
10    console.log("Variável local de função:", localFunctionVariable); // Ac
11    esso permitido à variável local de função dentro da função
12
13    // Bloco de código
14    if (true) {
15        // Variável local de bloco de código
16        let blockVariable = "Local de Bloco";
17        console.log("Dentro do bloco de código:");
18        console.log("Variável global:", globalVariable); // Acesso permiti
19    do à variável global dentro do bloco de código
20        console.log("Variável local de função:", localFunctionVariable); /\
21    / Acesso permitido à variável local de função dentro do bloco de código
22        console.log("Variável local de bloco de código:", blockVariable); \
23    // Acesso permitido à variável local de bloco de código dentro do bloco de
24    código
25    }
26
27    // Acesso a variável local de bloco de código fora do bloco (gera erro)
28    console.log("Fora do bloco de código:");
29    console.log("Variável local de bloco de código:", blockVariable); // I
30    sso causará um erro, pois blockVariable é uma variável local de bloco de c
31    ódigo e não está acessível fora dele
32 }
33
34 minhaFuncao();
35
36 // Acesso a variáveis globais fora da função (possível)
37 console.log("Fora da função:");
38 console.log("Variável global:", globalVariable); // Acesso permitido à var
39 iável global fora da função
40
41 // Acesso a variável local de função fora da função (gera erro)
42 console.log("Variável local de função:", localFunctionVariable); // Isso c\
```

```
43 ausará um erro, pois localFunctionVariable é uma variável local de função  
44 e não está acessível fora dela
```

Este exemplo demonstra como as variáveis globais podem ser acessadas em qualquer lugar do código, as variáveis locais de função só podem ser acessadas dentro da função em que foram declaradas, e as variáveis locais de bloco de código só podem ser acessadas dentro do bloco em que foram declaradas.

Em JavaScript, assim como na maioria das linguagens de programação, o escopo é delimitado por chaves, o que significa que as variáveis declaradas dentro de um bloco de código só são acessíveis dentro desse próprio bloco.

Entender o escopo é fundamental para organizar nosso código e evitar erros, pois nos ajuda a controlar onde cada parte do código pode acessar e modificar as variáveis.

## 6.6. Conclusão

Neste capítulo, exploramos detalhadamente o conceito de funções, desde sua declaração e chamada até o uso de parâmetros, retorno e escopos. Aprendemos sobre as funções predefinidas em JavaScript, que oferecem funcionalidades básicas e comuns prontas para uso, como entrada e saída de dados e manipulação de informações.

Compreendemos que as funções são blocos de código identificados por um nome único, utilizados para organizar, reutilizar e modularizar o código, tornando os programas mais eficientes e fáceis de manter. Além disso, exploramos a declaração de funções, tanto as tradicionais quanto as *arrow functions*, e vimos como realizar a execução de funções, passando argumentos e lidando com escopos de variáveis.

Também abordamos os diferentes tipos de escopos em JavaScript, global e local, que definem a visibilidade e acessibilidade das variáveis dentro do código.

Em suma, este capítulo proporcionou uma compreensão sólida das funções e sua importância na construção de aplicativos modernos. Com este conhecimento, os programadores estão mais capacitados para desenvolver código organizado, eficiente e fácil de manter, contribuindo para a excelência na prática da programação.

## 6.7. Exercícios

1. Liste as funções predefinidas em JavaScript que você conhece e categorize-as por funcionalidade (por exemplo, entrada e saída de dados, manipulação de strings, etc.).
2. Explore outras funções predefinidas em JavaScript lendo a documentação oficial e experimentando-as em seu próprio código.
3. Crie um código JavaScript que utilize pelo menos três funções predefinidas diferentes para realizar uma tarefa específica (por exemplo, calcular a média de três números).
4. Escreva uma função simples que exiba uma mensagem na tela e chame-a para ver o resultado.
5. Crie uma função que receba três números como parâmetros, calcule a média deles e exiba o resultado.
6. Escreva uma função que receba um número como parâmetro e retorne o quadrado desse número.
7. Declare uma variável global e uma local dentro de uma função. Tente acessá-las de diferentes partes do código para entender o escopo.
8. Crie uma função que declare uma variável dentro de um bloco de código (por exemplo, um `if`) e tente acessá-la fora desse bloco para entender o escopo local do bloco.
9. Reescreva todas as funções anteriores usando a estrutura de *arrow function*.
10. Crie um pequeno programa que solicite dois números ao usuário, calcule sua média e exiba o resultado usando funções.
11. Crie um jogo em que o programa escolha um número aleatório e solicite ao usuário que tente adivinhá-lo. Use uma função para gerar o número aleatório e outra função para verificar se a suposição do usuário está correta.
12. Crie uma calculadora que solicite ao usuário que escolha uma operação (adição, subtração, multiplicação ou divisão) e dois números como entrada. Use funções separadas para cada operação e exiba o resultado.
13. Desenvolva um conversor de moeda que solicite ao usuário o valor em uma moeda e a taxa de conversão para outra moeda. Use uma função para realizar a conversão e exibir o resultado.

14. Escreva uma função que solicite ao usuário sua altura e peso e calcule o IMC. Exiba o resultado e uma mensagem indicando se a pessoa está abaixo do peso, dentro do peso normal, com sobrepeso ou obesa.

# 7. Estruturas de dados básicas

## 7.1. Estruturas de dados

Uma estrutura de dados é uma forma de organizar e armazenar dados em um computador de maneira eficiente. Ela define como os dados são armazenados na memória e como podem ser acessados e manipulados por programas.

Ao escolher a estrutura de dados adequada para um determinado problema ou aplicação, os programadores podem otimizar o uso de memória, acelerar a execução de algoritmos e simplificar a manipulação e organização dos dados. Por exemplo, ao lidar com grandes conjuntos de dados, escolher a estrutura de dados certa pode fazer a diferença entre um programa que funciona rapidamente e um que é lento e ineficiente.

Além disso, as estruturas de dados desempenham um papel crucial na organização e modelagem de informações em sistemas de software complexos. Elas permitem representar de forma eficaz relacionamentos entre diferentes tipos de dados e facilitam a implementação de algoritmos e operações específicas.

Compreender as estruturas de dados e saber como aplicá-las corretamente é fundamental para qualquer programador, independentemente do nível de experiência. Ao dominar as diferentes estruturas de dados disponíveis e entender seus prós e contras, os desenvolvedores podem escrever código mais eficiente, escalável e fácil de manter, contribuindo para o sucesso e a qualidade dos projetos de software.

Existem várias estruturas de dados para organizar e armazenar informações na memória do computador. Desde as mais simples, como matrizes, conjuntos, mapas, filas e pilhas, até as mais complexas, como árvores e grafos, cada uma possui suas características únicas e aplicações específicas. Conhecer essas estruturas é essencial para desenvolver software eficiente e robusto. Neste livro, exploraremos em detalhes três estruturas de dados básicas: matriz (array em inglês), conjunto (set em inglês) e mapa (map em inglês), utilizando os termos em inglês, amplamente reconhecidos no mundo da programação.

## 7.2. Arrays

Arrays, também conhecidos como vetores ou matrizes em português, são estruturas de dados fundamentais em programação, utilizadas para armazenar coleções de elementos organizados de forma sequencial na memória do computador. Eles oferecem uma maneira eficiente de gerenciar conjuntos de dados relacionados, permitindo acesso rápido e manipulação flexível dos elementos.

Em JavaScript, arrays são estruturas de dados que permitem armazenar **coleções de elementos de tipos diferentes**, como números, strings, objetos e até mesmo outros arrays. Além disso, os arrays em JavaScript são dinâmicos, o que significa que podem crescer ou encolher conforme necessário.

As principais características dos arrays em JavaScript são:

- **Flexibilidade de tipos de dados:** Um pode conter elementos de diferentes tipos de dados, incluindo números, strings, objetos, funções e até mesmo outros arrays.
- **Tamanho dinâmico:** Ao contrário de algumas outras linguagens de programação, como Java, C e C# por exemplo, os arrays em JavaScript têm um tamanho dinâmico. Isso significa que você pode adicionar ou remover elementos do array conforme necessário, sem a necessidade de especificar um tamanho fixo durante a sua declaração.
- **Acesso por índice:** Os elementos em um array são acessados por meio de um índice numérico. O primeiro elemento tem o índice 0, o segundo tem o índice 1 e assim por diante. Isso permite uma rápida e eficiente recuperação de elementos específicos do array.
- **Métodos de Manipulação:** JavaScript fornece uma variedade de métodos embutidos para manipulação de arrays, como `push()`, `pop()`, `shift()`, `unshift()`, `splice()`, `concat()`, `slice()` e muitos outros. Esses métodos permitem adicionar, remover, modificar e acessar elementos do array de maneira fácil e eficiente.
- **Iteração:** Arrays podem ser facilmente percorridos usando loops como `for`. Isso facilita a execução de operações em cada elemento do array.
- **Passagem por referência:** Em JavaScript, os arrays são passados por referência, o que significa que quando você atribui um array a uma variável ou passa essa variável como argumento para uma função, na verdade você

está passando uma referência à localização na memória onde o array está armazenado. Assim, qualquer modificação efetuada no array dentro da função refletirá também fora dela e vice-versa.

## 7.2.1. Declaração e inicialização de arrays

Existem duas maneiras principais de declarar um array em JavaScript: Declaração literal e usando método construtor.

Na declaração literal, o array é definido utilizando colchetes [], e os elementos são separados por vírgulas. Veja os exemplos abaixo:

```
1 // Array com elementos do mesmo tipo: string
2 const frutas = ["Maçã", "Banana", "Laranja"];
3 // Array com elementos do mesmo tipo: número
4 const numeros = [1, 2, 3, 4, 5];
5 // Array com elementos de tipos diferentes: string, número e booleano
6 const misturado = ["Olá", 10, true];
```

Também é possível usar o método construtor Array para criar um array. Veja os exemplos a seguir:

```
1 const frutas = new Array("Maçã", "Banana", "Laranja");
2 const numeros = new Array(1, 2, 3, 4, 5);
3 const misturado = new Array("Olá", 10, true);
```

Nos exemplos anteriores, os arrays foram declarados e inicializados com valores específicos. No entanto, é possível declarar arrays vazios. Veja abaixo exemplos de declaração de arrays vazios:

```
1 const frutas = [];
2 const numeros = new Array();
3 const misturado = new Array(4); // cria um array vazio com 4 elementos
```



## 7.2.2. Acesso e manipulação de elementos

É possível acessar elementos específicos de um array usando sua posição, que é representada por um índice numérico. O índice do primeiro elemento é 0, o do segundo é 1 e assim por diante.

Em JavaScript, você pode acessar e manipular elementos de arrays de várias maneiras. Para acessar um elemento específico em um array, você usa a notação de colchetes [], especificando o índice do elemento desejado. Por exemplo, `array[índice]` retorna o elemento no índice especificado do array. Veja o exemplo abaixo:

```
1  const frutas = ["Maçã", "Banana", "Laranja"];
2
3  // Acessando o primeiro elemento (índice 0)
4  const primeiraFruta = frutas[0];
5  console.log(primeiraFruta) // "Maçã"
6
7  // Acessando um elemento específico
8  const segundaFruta = frutas[1];
9  console.log(segundaFruta) // "Banana"
```

Além de acessar, é possível substituir um elemento de um determinado índice, como por exemplo:

```
1  const frutas = ["Maçã", "Banana", "Laranja"];
2
3  // Modificando o valor do segundo elemento
4  frutas[1] = "Morango";
5
6  console.log(frutas); // ["Maçã", "Morango", "Laranja"]
```

Para realizar operações de manipulação em arrays, como adicionar, remover, modificar e pesquisar elementos, utilizamos métodos específicos predefinidos em JavaScript. Alguns desses métodos comuns são:

- **push()**: adiciona um ou mais elementos ao final do array.
- **pop()**: remove o último elemento do array.
- **shift()**: remove o primeiro elemento do array.

- **unshift()**: adiciona um ou mais elementos ao início do array.
- **slice()**: retorna uma cópia de uma parte do array.
- **concat()**: concatena dois ou mais arrays em um único array.
- **sort()**: ordena os elementos do array em ordem crescente ou decrescente.

O termo “método” é usado para descrever uma função que está associada a um objeto específico em programação orientada a objetos (Capítulo 9). Em JavaScript, geralmente chamamos tanto as funções independentes quanto aquelas associadas a objetos de “métodos”. Basicamente não há diferença entre uma função e um método e ambos os termos são frequentemente usados de forma intercambiável.

A seguir serão listados alguns exemplos de código e explicações para cada uma das funções mencionadas acima:

## 8. Modularização

Em construção...

# 9. JavaScript Orientado a Objetos

Em construção...

# 10. Testes Automatizados

Em construção...

# 11. Introdução a TypeScript

Em construção...