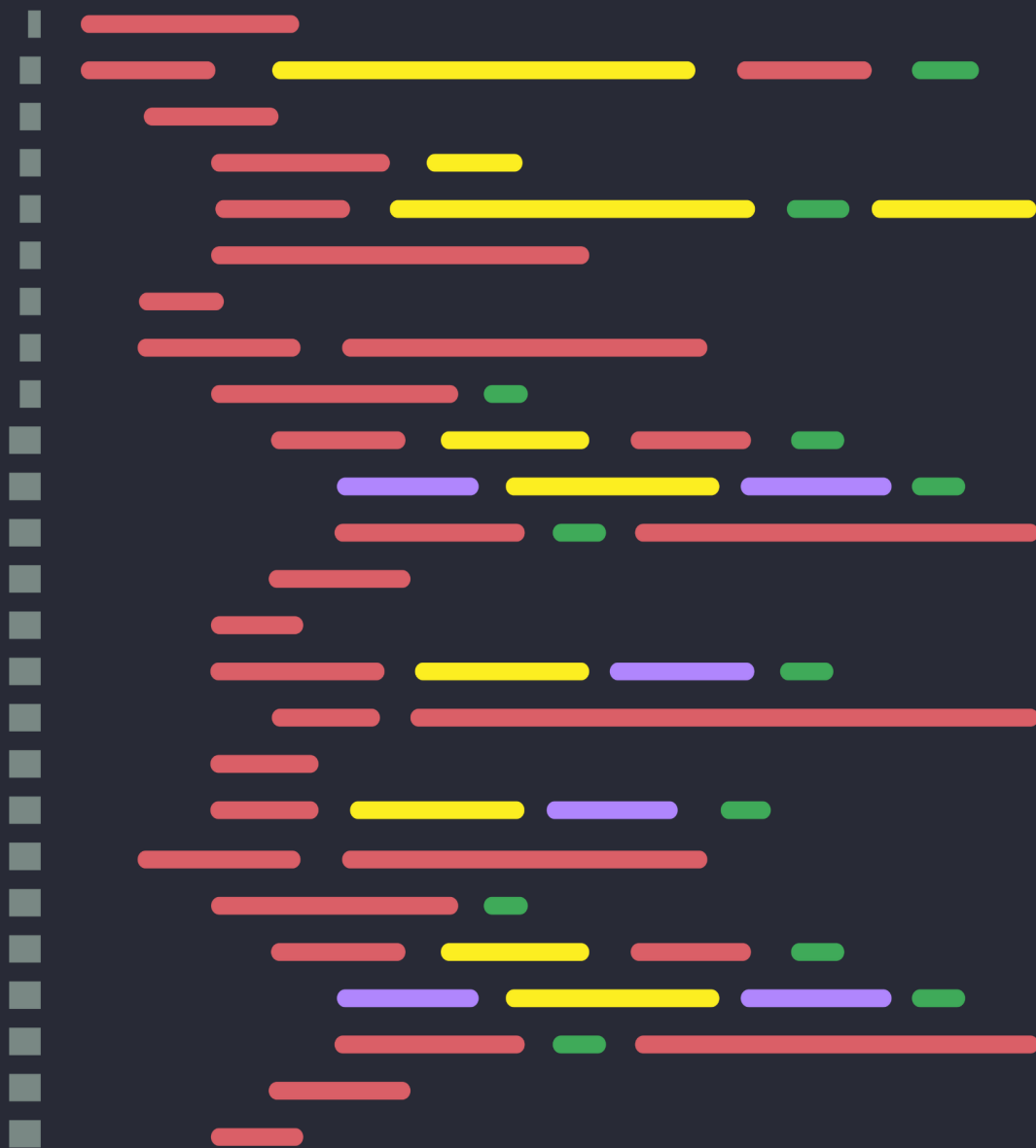


Aprenda Programar com JavaScript



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 2024-12-23 ISBN 978-65-01-03426-3



Prefácio

Este livro foi desenvolvido com o propósito de proporcionar uma jornada de aprendizado descomplicada e eficiente em programação. Cada capítulo foi cuidadosamente planejado para atender às necessidades atuais de ensino e aprendizado, apresentando uma abordagem prática e acessível. O conteúdo abrange desde os fundamentos da programação até tópicos mais avançados, como programação orientada a objetos e testes automatizados, sempre com o objetivo de facilitar a compreensão e a aplicação dos conceitos aprendidos.

Para enriquecer o conteúdo, o autor utilizou tecnologias de inteligência artificial, como o GPT da OpenAI e o Gemini do Google, para aperfeiçoar partes do texto. Após a utilização dessas ferramentas, o autor realizou uma análise, edição e revisão linguística criteriosa, assumindo a responsabilidade final pelo conteúdo desta publicação.

“Aprenda Programar com JavaScript” é ideal para iniciantes que desejam ingressar na programação e para professores que buscam material didático com conteúdo acessível e exercícios práticos para enriquecer suas aulas.

Os professores que adotarem este livro em suas disciplinas podem solicitar uma cópia em PDF para distribuir aos seus alunos. Para isso, basta entrar em contato com o autor: @jesielviana

Autor

Jesiel Viana é um profissional com mais de 15 anos de experiência em desenvolvimento de software e ensino de programação. Especialista em desenvolvimento web, possui vasta experiência em tecnologias como Java, JavaScript, Node, Angular, Vue e React. Mestre em Engenharia de Software, Jesiel é atualmente professor efetivo no Instituto Federal do Piauí (IFPI) e colaborador em projetos de pesquisa no Instituto Brasileiro de Informação em Ciência e Tecnologia (IBICT).

© 2024 Jesiel Viana

Dedico este livro aos meus pais, cujo apoio incansável aos meus estudos foi fundamental para minha jornada. À minha esposa Mayanny e minhas duas filhas (Melissa e Jade), 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

1. Introdução à Programação	1
1.1. O que é programação?	1
1.2. História da programação	1
1.3. Importância da programação no contexto atual	3
1.4. Similaridade entre linguagens de programação	3
1.5. Por que JavaScript é a linguagem ideal para iniciantes?	4
1.6. Conclusão	5
2. Lógica de Programação e Algoritmos	6
2.1. Lógica de programação	6
2.2. Algoritmos	6
2.3. Pensamento computacional	10
2.4. Conclusão	11
2.5. Exercícios	12
3. Introdução ao JavaScript	13
3.1. Principais características	13
3.2. Surgimento e evolução	14
3.3. Ambientes de execução	15
3.4. Editor de código	17
3.5. Saída de dados com console.log()	18
3.6. Conclusão	19
3.7. Exercícios	19
4. Tipos de Dados e Variáveis	21
4.1. Tipos de Dados	21
4.2. Variáveis	22
4.3. Sintaxe básica do JavaScript	26
4.4. Entrada de dados com prompt()	28
4.5. Conversão de tipos de dados	29
4.6. Conclusão	31
4.7. Exercícios resolvidos	31
4.8. Exercícios	32
5. Operadores e Estruturas de Controle de Fluxo	33
5.1. Operadores	33
5.2. Controle de fluxo	39
5.3. Conclusão	50

5.4. Exercícios resolvidos	50
5.5. Exercícios	52
6. Funções e Escopo	54
6.1. Funções	54
6.2. Declaração de funções	54
6.3. Execução de funções	57
6.4. A importância do uso de funções	59
6.5. Funções predefinidas	59
6.6. Escopo	60
6.7. Conclusão	62
6.8. Exercícios	62
7. Objetos, Arrays e Strings	65
7.1. O que são estruturas de dados?	65
7.2. Objetos	65
7.3. Arrays	69
7.4. Strings	79
7.5. Conclusão	83
7.6. Exercícios	83
8. Modularização	87
8.1. Modularização	87
8.2. CommonJS	88
8.3. Módulos JavaScript	89
8.4. Módulos externos	91
8.5. Conclusão	92
8.6. Exercícios	93
9. Programação Orientada a Objetos com JavaScript	94
9.1. Programação Orientada a Objetos	94
9.2. POO com JavaScript	95
9.3. Conclusão	101
9.4. Exercícios	101
10. Testes Automatizados	103
10.1. Introdução	103
10.2. Tipos de testes automatizados	104
10.3. Ferramentas para automação de testes em JavaScript	104
10.4. Testes automatizados na prática	105
10.5. Conclusão	110
10.6. Exercícios	111
11. Introdução ao TypeScript	112
11.1. O que é TypeScript?	112
11.2. Declaração de variáveis	113
11.3. Declaração de funções	114

11.4. Interfaces	114
11.5. Classes e modificadores de acesso	116
11.6. Conclusão	118
11.7. Exercícios	118
12. Para Professores	120
12.1. E-book grátis para alunos	120
13. Conclusão	121

1. Introdução à Programação

A programação de computadores é uma habilidade poderosa no mundo digital, permeando diversas esferas do cotidiano, desde o entretenimento até os processos empresariais. Com o aumento da demanda por soluções tecnológicas inovadoras e a transformação digital de diversos setores, aprender a programar tornou-se uma necessidade para quem deseja se destacar no mercado de trabalho e compreender o funcionamento dos sistemas que nos cercam.

Neste capítulo introdutório, vamos explorar os fundamentos da programação, desde suas origens históricas até sua importância no contexto atual. Veremos como as linguagens de programação evoluíram ao longo do tempo, refletindo as necessidades e demandas da sociedade em constante mudança. Por fim, discutiremos a convergência de recursos entre as linguagens de programação e por que JavaScript emerge como uma escolha ideal para quem está começando nessa jornada.

1.1. O que é programação?

Programação é a ferramenta que utilizamos para instruir o computador a executar tarefas específicas. Consiste em escrever um conjunto de instruções detalhadas, conhecidas como código-fonte, que descrevem passo a passo o que o computador deve fazer.

Para escrever essas instruções, utilizamos uma linguagem de programação, que funciona como um idioma de comunicação entre o programador e o computador. As linguagens de programação são compreensíveis tanto para nós, humanos, quanto para os computadores.

Há uma variedade de linguagens de programação disponíveis, cada uma com sua própria sintaxe e características específicas. Assim como no mundo das línguas naturais, podemos escolher a linguagem mais adequada para “conversar” com os computadores e desenvolver os programas desejados.

1.2. 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:

1. **Século XIX (1800):** Ada Lovelace é reconhecida como a primeira programadora ao escrever o primeiro algoritmo para ser processado por um dispositivo, 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.

2. **Décadas de 1930 e 1940:** Alan Turing desenvolveu 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), o primeiro computador digital eletrônico de grande escala.
3. **Anos 1950:** emergem as primeiras linguagens de programação de alto nível, como Fortran e COBOL. Grace Hopper desenvolve o primeiro compilador.
4. **Década de 1970:** surgem linguagens de programação mais acessíveis, como C, Pascal e Smalltalk, que facilitam a escrita de programas e impulsionam a popularização da programação. A programação orientada a objetos ganha destaque com o Smalltalk. O desenvolvimento de microcomputadores, como o Altair 8800 e o Apple II, leva a computação para os lares, estimulando o interesse pela programação entre o público em geral.
5. **Década de 1980:** a explosão dos computadores pessoais, com o IBM PC e o Macintosh, democratizou o acesso à tecnologia e criou um mercado em massa para softwares. As linguagens de programação evoluíram, com o surgimento de C++ e Objective-C, que combinam características de linguagens anteriores. O desenvolvimento de interfaces gráficas amigáveis, como as do Windows e do Mac OS, tornou os computadores mais intuitivos e acessíveis.
6. **Década de 1990:** o surgimento da Web, criada por Tim Berners-Lee, transformou a comunicação e o acesso à informação. A internet cresceu rapidamente, com o desenvolvimento de navegadores web que facilitaram a navegação. Novas linguagens de programação, como Java, JavaScript e Python, foram criadas para atender às necessidades da web. Essa década também viu a popularização dos jogos online e o crescimento do comércio eletrônico.
7. **Década de 2000 em diante:** a ascensão das redes sociais, a popularização dos smartphones e o avanço das técnicas de inteligência artificial revolucionaram vários setores da sociedade.

No século XIX, não havia computadores, mas as tecnologias da época estabeleceram princípios fundamentais que influenciaram a programação moderna. Conceitos como automação, lógica computacional e armazenamento de instruções em cartões perfurados prepararam o caminho para o desenvolvimento 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 essenciais para o desenvolvimento da programação moderna. Esses avanços estabeleceram as bases para a criação de softwares mais complexos e poderosos. As linguagens, ferramentas e conceitos desenvolvidos nesse período continuam a ser utilizados, influenciando a forma como criamos e utilizamos softwares em diversos campos até hoje.

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 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.

1.3. Importância da programação no contexto atual

Aprender programação de computadores é uma habilidade essencial, dado o papel central que a programação desempenha na sociedade atual. Ela abre inúmeras oportunidades e é uma competência relevante em diversos setores. Aqui estão algumas razões importantes para aprender a programar:

1. **Automação:** a programação é a chave para a automação de processos, desde linhas de produção industriais até tarefas do dia a dia, proporcionando eficiência e redução de erros.
2. **Desenvolvimento de aplicativos:** a programação é fundamental para criar soluções que atendem às diversas necessidades dos usuários, desde aplicativos móveis e aplicações web até plataformas complexas de gerenciamento empresarial.
3. **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.
4. **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.
5. **Capacitação profissional:** a programação é uma competência valiosa na formação educacional e no mercado de trabalho, preparando indivíduos para enfrentar desafios em diversas áreas e setores.
6. **Empregabilidade:** há uma demanda crescente por profissionais de TI e desenvolvedores de software em diversos setores. Aprender programação aumenta suas oportunidades de emprego.
7. **Autonomia digital:** a programação oferece autonomia ao permitir que você crie suas próprias ferramentas, websites ou soluções para desafios específicos.

1.4. Similaridade entre linguagens de programação

No mundo da programação, a diversidade de linguagens pode parecer assustadora à primeira vista. No entanto, assim como as línguas naturais compartilham conceitos básicos como substantivos, verbos e frases, as linguagens de programação também apresentam fundamentos em comum. Dominar esses fundamentos é a chave para desvendar as particularidades de cada linguagem e se tornar um bom programador.

Embora a sintaxe, ou seja, a maneira como os conceitos são escritos, varie entre as linguagens, os pilares da programação permanecem os mesmos. Algoritmos, estruturas de dados, variáveis, tipos de dados, operadores e controle de fluxo são alguns exemplos desses fundamentos. Dominá-los permite que você compreenda a lógica por trás da programação, independentemente da linguagem escolhida.

Ao começar sua jornada na programação, não se intimide com a variedade de linguagens disponíveis e evite tentar aprender várias ao mesmo tempo. Concentre-se em aprender uma linguagem e dominá-la. Esse conhecimento sólido facilitará o aprendizado de outras linguagens no futuro, pois você já estará familiarizado com os conceitos fundamentais.

Aprender os fundamentos da programação é o primeiro passo para se tornar um programador excelente. A jornada de aprendizado em programação é desafiadora, requer dedicação e prática, mas também é muito gratificante.

1.5. Por que JavaScript é a linguagem ideal para iniciantes?

Ao escolher a primeira linguagem de programação para aprender, é essencial considerar alguns fatores que podem influenciar sua experiência e progresso.

Embora muitas linguagens compartilham semelhanças e recursos comuns, alguns critérios são importantes avaliar nesse processo. A busca por uma linguagem acessível, fácil de usar e com sintaxe amigável é fundamental para iniciar sua jornada com sucesso.

Nesse contexto, o JavaScript se destaca como uma excelente opção para iniciantes. Sua presença ampla na web e em diversas ferramentas facilita o acesso e remove barreiras iniciais. A sintaxe clara e intuitiva do JavaScript torna a compreensão e a escrita do código mais simples, fazendo com que o aprendizado seja mais agradável.

Vantagens do JavaScript para iniciantes:

- **Acessibilidade:** JavaScript está presente em navegadores web, permitindo que você execute seus primeiros programas sem precisar instalar softwares complexos. Isso facilita o aprendizado e a experimentação, pois você pode testar seu código instantaneamente em qualquer navegador.

- **Sintaxe simples:** a sintaxe do JavaScript é acessível, simples, concisa e intuitiva, permitindo que se concentre nos conceitos fundamentais sem complicações desnecessárias.
- **Versatilidade:** o JavaScript vai além da web. Ele pode ser utilizado para desenvolver aplicativos móveis, jogos, interfaces gráficas e até mesmo aplicações de servidores. Essa versatilidade garante que você possa aplicar seus conhecimentos em diversos projetos e áreas da programação.
- **Comunidade e recursos disponíveis:** a comunidade JavaScript é uma das maiores e mais ativas do mundo. Isso significa que você terá acesso a uma grande quantidade de recursos de aprendizado grátis, tutoriais, fóruns e comunidades online para tirar suas dúvidas e receber ajuda.

Este livro foca nos fundamentos da programação, explorando recursos comuns a todas as linguagens. Devido à semelhança entre elas, uma vez aprendidos em JavaScript, é fácil aplicar esses conceitos em outras linguagens, precisando apenas consultar a sintaxe correspondente.

1.6. Conclusão

Neste capítulo, exploramos a importância e a evolução da programação, desde suas raízes históricas até sua relevância no mundo atual. Vimos como a programação se tornou uma habilidade essencial, impulsionando inovações tecnológicas e oferecendo inúmeras oportunidades no mercado de trabalho. Também discutimos a convergência de recursos entre diferentes linguagens de programação, destacando como o conhecimento dos fundamentos pode facilitar a transição entre elas.

Destacamos o JavaScript como uma linguagem ideal para iniciantes, devido à sua ampla acessibilidade, sintaxe simples, versatilidade e à rica comunidade de recursos disponíveis. Ao dominar o JavaScript, você estará bem preparado para explorar outras linguagens de programação.

Nos próximos capítulos, vamos explorar desde os conceitos básicos até os mais avançados da programação, aplicando o que aprendemos em projetos práticos com JavaScript. Este é apenas o começo da sua jornada emocionante na programação. Desejo boa sorte e bons estudos!

2. Lógica de Programação e Algoritmos

Este capítulo apresenta os conceitos fundamentais da programação: lógica, algoritmos e pensamento computacional. O domínio desses elementos é essencial para quem deseja iniciar sua jornada na programação. Antes de escrever qualquer linha de código, é fundamental desenvolver a capacidade de pensar de forma estruturada para solucionar problemas de forma eficiente.

2.1. Lógica de programação

Lógica de programação é a capacidade de criar sequências lógicas de instruções para um computador realizar 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 iterativa 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, será apresentado 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, sendo:

- 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 ser 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:

```
1  Início do algoritmo.
2  Solicitar ao usuário que forneça o primeiro número.
3  Armazenar o primeiro número fornecido.
4  Solicitar ao usuário que forneça o segundo número.
5  Armazenar o segundo número fornecido.
6  Somar os dois números.
7  Armazenar o resultado da soma.
8  Calcular a média dividindo a soma por 2.
9  Armazenar o resultado da média.
10 Exibir a média calculada ao usuário.
11 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 pseudocódigo, a qual é uma representação intermediária 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 pseudo linguagem auxilia no aprendizado de conceitos básicos de algoritmos e lógica de programação, utilizando uma sintaxe próxima do português. Ela é geralmente adotada como uma etapa inicial antes do estudo de uma linguagem de programação real.

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, tornando o código mais acessível para falantes nativos dessa língua.

- **Ambiente de desenvolvimento:** minimalista, com uma interface simples e 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.

```
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 a sintaxe de uma pseudo linguagem, 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. O quadro a seguir apresenta a simbologia de fluxograma, normatizada pela ISO 5807, utilizada para descrever algoritmos.

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 as etapas

A Figura abaixo mostra o fluxograma do algoritmo para calcular a média de dois números, utilizando a simbologia apresentada acima.

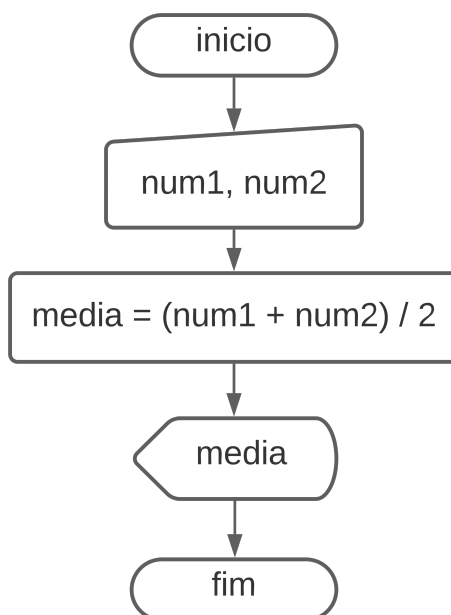


Figura 2.1. Fluxograma do algoritmo que calcula a média de dois números

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

2.3. Pensamento computacional

O pensamento computacional é uma abordagem sistemática para resolver problemas de forma lógica, que envolve a decomposição de tarefas complexas em partes menores e mais

gerenciáveis. Essa metodologia é aplicável em diversas áreas, não apenas na computação, e se sustenta em quatro pilares principais:

- **Decomposição:** Refere-se à divisão de um problema complexo em componentes menores, permitindo que cada subproblema seja resolvido de maneira isolada. Isso facilita a compreensão e a solução do problema de forma global.
- **Reconhecimento de padrões:** Consiste em identificar semelhanças ou regularidades entre problemas ou dentro de um mesmo problema, permitindo a aplicação de soluções já existentes em novos contextos e otimizando o processo de resolução.
- **Abstração:** Envolve a eliminação de detalhes irrelevantes, concentrando-se apenas nos aspectos essenciais de um problema. A abstração simplifica a complexidade, criando representações ou modelos que capturam a essência do problema sem se perder em particularidades.
- **Algoritmo:** Trata-se de uma sequência de passos lógicos e ordenados que visa resolver um problema específico. Um algoritmo deve ser claro, eficiente e repetível, podendo ser descrito em linguagem comum ou implementado em código.

A lógica desempenha um papel crucial ao conectar e organizar esses pilares, permitindo o desenvolvimento de soluções estruturadas, eficientes e reutilizáveis quando aplicados em conjunto.

Desenvolver o pensamento computacional é fundamental para resolver problemas de forma lógica e estruturada, que são a base da programação.

2.4. Conclusão

A lógica de programação é o pensamento abstrato que vem antes da criação de algoritmos, os quais são a aplicação prática dessa lógica, consistindo em sequências 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 representação 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, por possibilitar 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.

Por fim, o pensamento computacional oferece uma abordagem estruturada para decompor problemas complexos, identificar padrões, abstrair informações irrelevantes e construir soluções com base em algoritmos. Essa prática vai além da programação, pois fortalece a capacidade de pensar de forma analítica e lógica, aplicando estratégias eficientes em problemas de diversas áreas.

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.

2.5. Exercícios

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

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

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.

Começamos 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 diversas linguagens de programação, JavaScript é uma das melhores opções para iniciantes. Sua sintaxe é simples e fácil de entender, o que torna o aprendizado mais acessível para iniciantes. Outra vantagem é que o JavaScript pode ser executado diretamente em qualquer navegador web, sem configurações complicadas, facilitando 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 específicos de dados.
- **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 dinâmico. Permite a depuração de erros de forma mais fácil.
- **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:** o 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 está em 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/)¹ e [StackOverflow](https://survey.stackoverflow.co/)², 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 estão listados os principais acontecimentos na história do JavaScript.

¹<https://octoverse.github.com/>

²<https://survey.stackoverflow.co/>

- **1995:** a Netscape Communications Corporation, sob a liderança de Brendan Eich, cria o JavaScript. Inicialmente chamado de LiveScript, foi rapidamente 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.
- **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 baseada no padrão ECMAScript, que define suas características e funcionalidades.

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)³, [Deno](https://deno.com)⁴ e [Bun](https://bun.sh)⁵, conhecidos como plataformas de execução de JavaScript do lado do servidor. O Node.js foi pioneiro e é amplamente utilizado.

³<https://nodejs.org>

⁴<https://deno.com>

⁵<https://bun.sh>

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.

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 possuem um mecanismo JavaScript embutido. **Utilize o console do navegador somente para testar e depurar seu código.**

3.3.2. Bun

Bun é uma plataforma para execução de JavaScript com foco no desempenho e em uma experiência de desenvolvimento simplificada.

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.

Outra vantagem é que 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 moderno sem a necessidade de fazer alterações.

Para executar código JavaScript localmente em seu computador usando o Bun, é necessário realizar sua instalação.

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

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

Para instalar no Windows, basta executar o seguinte comando no PowerShell/cmd.exe do seu computador:

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

Depois que o comando for processado, verifique se a instalação foi bem-sucedida executando:

```
1 bun -v
```

Esse comando 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, podemos facilmente executar arquivos JavaScript diretamente pelo terminal do computador ou pelo terminal integrado da ferramenta de codificação, o que é bastante conveniente. Basta digitar no terminal o comando “bun nomeArquivo.js”.

3.4. Editor de código

Para programar em JavaScript, recomendo o *Visual Studio Code (VS Code)*, um editor gratuito que reúne todos os recursos necessários para o desenvolvimento e se destaca como o mais utilizado no mercado.

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 uma das principais escolhas dos desenvolvedores são:

- **Gratuito e Open Source:** é um software livre para uso, e seu código-fonte está acessível para modificação e distribuição por qualquer pessoa.
- **Multiplataforma:** compatível com Windows, macOS e Linux, permitindo que você o utilize em qualquer sistema operacional.
- **Leve e Rápido:** um editor de código leve e rápido, garantindo que ele não sobrecarregue seu computador durante o uso.
- **Interface Personalizável:** sua interface 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 e recomendo o VS Code. 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.

Sugestão: pesquise extensões do VS Code para ampliar as funcionalidades e otimizar sua experiência de desenvolvimento com JavaScript e Bun. Personalize sua interface conforme suas preferências e necessidades específicas.

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 exibir mensagens diretamente no console do navegador ou no terminal de desenvolvimento. Essa instrução é geralmente utilizada por desenvolvedores para depurar o programa durante a sua execução.

Depuração é o processo de análise de código para localizar e corrigir erros em um programa de computador.

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, conhecemos 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.

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, é uma alternativa moderna ao Node.js, com instalação simples e sem a necessidade de configurações iniciais, tornando-o ideal para iniciantes.

Também exploramos o editor de código Visual Studio Code (VS Code), que oferece um ambiente de desenvolvimento completo. O VS Code se destaca por ser uma IDE robusta, altamente personalizável e amplamente utilizada.

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 de 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. Explique a função da instrução `console.log()` em JavaScript.
6. Utilize o console do navegador para exibir a mensagem “Olá mundo!” usando o `console.log()`.
7. Crie um projeto JavaScript no *VS Code* que imprima a mensagem “Olá mundo!” no terminal usando `console.log()`.
8. Pesquise sobre extensões do *VS Code* para adicionar funcionalidades e aprimorar sua experiência de desenvolvimento com JavaScript e Bun.
9. Personalize a interface 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.

Adicionalmente, 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, abordaremos 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 (verdade 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, organizados 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, sendo utilizado para representar diferentes tipos de informações em 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 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`: utilizada para declarar variáveis cujo valor não pode ser reatribuído, garantindo que ele permaneça constante..

Veja abaixo exemplos de declarações de variáveis:

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

- `const`: é a palavra-chave usada para declarar uma constante.
- `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 identificador = 'valor';`.

Nesse caso, `identificador` é o nome dado à variável, enquanto `'valor'` é o conteúdo que será armazenado nela.

Variáveis declaradas com `let` podem ser inicializadas na própria declaração ou em qualquer momento posterior no código. Ao declarar uma variável, você pode atribuir um valor a ela imediatamente. Por exemplo:

```
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 declarar uma variável com `const`, é obrigatório atribuir um valor imediatamente, pois variáveis “const” são constantes e não podem ser reatribuídas. Portanto, a inicialização deve ocorrer na própria declaração. Veja o exemplo:

```
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 for necessário.

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 incluindo 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 serve 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 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 `anoDePublicacao` é inicializada com o número 2024. Isso significa que o tipo da variável `anoDePublicacao` é inferido como um número inteiro.

É possível verificar o tipo de uma variável usando o operador `typeof` como demonstrado 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 indica o tipo da variável. No exemplo acima, `typeof livro` resulta em “string”, enquanto `typeof anoDePublicacao` resulta em “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 usaremos 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. Veja o exemplo de código sem ponto e vírgula:

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

Mesmo o ponto e vírgula em JavaScript não seja obrigatório, ele é útil para evitar ambiguidade e melhorar a legibilidade do código. Neste livro usaremos ponto e vírgula como convenção.

4.3.4. Comentários

Comentários em JavaScript são trechos de texto 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:** começa com `//` e termina no final da linha.
- **Comentários de várias linhas:** inicia com `/*` e termina 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
5 linhas. */
6
7 // Declaração e inicialização da variável 'mensagem'.
8 const mensagem = 'Venha aprender programar com o livro Aprenda a Programar com Ja\
9 vaScript.';
10
11 // Esta linha imprime o valor da variável 'mensagem'.
12 console.log(mensagem); /* Saída: Venha aprender programar com o livro Aprenda a P\
13 rogramar com JavaScript */
```

Os comentários são úteis para melhorar a legibilidade, a manutenção e a depuração do código. Use-os com moderação para garantir sua eficácia. Evite poluir o código com comentários desnecessários.

4.4. Entrada de dados com prompt()

O `prompt()` é uma função JavaScript que solicita ao usuário a entrada de dados e recebe o valor digitado. É uma função útil para interações básicas em programas simples, permitindo a coleta de dados para uso dinâmico.

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 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)¹, que funciona de forma similar ao `prompt()` do navegador e do Bun.

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

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 na 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:** é 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:** é efetuada 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.

```
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 valor da variável `num` para uma string e, em seguida, realiza 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 adequadamente 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 oferece flexibilidade no manejo de diferentes tipos de dados, enquanto o uso consistente de aspas, ponto e vírgula e comentários são elementos essenciais da sintaxe que ajudam a estruturar e manter 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 JavaScript e desenvolver soluções eficientes e robustas com ela. **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, exibe o tipo de dado inserido e, em seguida, converte essa entrada para um número usando `Number()` e por fim, o tipo de dado convertido é exibido no console.

```
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 na programação? Por que são necessárias?
2. O que é tipagem dinâmica em JavaScript? Como é que 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 divisão: 1 */
20
21 // Incremento
22 let incremento = 1;
23 incremento++;
24 console.log('incremento: ', incremento); // Saída => incremento: 2
25
26 // Decremento
27 let decremento = 2;
28 decremento--;
29 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
6 let quantidade = 20;
7 // Atribuição com adição
8 quantidade += 2;
9 console.log('quantidade:', quantidade); // Saída => quantidade: 22
10
11
12 let saldo = 100;
13 // Atribuição com subtração
14 saldo -= 50; // saldo = 50
15 console.log('saldo:', saldo); // Saída => saldo: 50
16
17
18 let pontuacao = 10;
19 // Atribuição com multiplicação
20 pontuacao *= 2;
21 console.log('pontuacao:', pontuacao); // Saída => pontuacao: 20
22
23
24 let tempo = 60;
25 // Atribuição com divisão
26 tempo /= 2; // tempo = 30
27 console.log('tempo:', tempo); // Saída => tempo: 30
28
29
30 let resultado = 10 % 3;
31 // Atribuição com resto da divisão
32 resultado %= 2;
33 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;
3 console.log('igualdade: ', igualdade); // Saída => igualdade: true
4
5 // Igualdade (compara valores)
6 igualdade = 2 == '2';
7 console.log('igualdade: ', igualdade); // Saída => igualdade: true
8
9 // Igualdade estrita (compara valores e tipos)
10 let igualdadeEstrita = 2 === 2;
11 console.log('igualdadeEstrita:', igualdadeEstrita); /* Saída => igualdadeEstrita:\
12 true */
13
14 // Igualdade estrita (compara valores e tipos)
15 igualdadeEstrita = 2 === '2';
16 console.log('igualdadeEstrita:', igualdadeEstrita); /* Saída => igualdadeEstrita:\
17 false */
18
19 // Diferença (compara se os 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 => diferencaEstrita:\
26 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;
19 console.log('negacao:', negacao); // Saída => negacao: true
```

Operadores lógicos permitem criar 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 espaço \
9 entre elas */
10 let nomeCompleto = nome + ' ' + sobrenome;
11 console.log('nomeCompleto:', nomeCompleto); /* Saída => nomeCompleto: Ana 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

O controle de fluxo é uma técnica essencial na programação que direciona a execução de um programa com base em condições ou critérios específicos. Ele permite tomar decisões lógicas durante a execução do código, determinando quais partes serão executadas com base em variáveis, valores de retorno de funções ou entradas do usuário.

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 executam um bloco de código repetidamente enquanto uma condição 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 primordiais para a programação. São essenciais em todos os programas, desde os simples até os complexos. São necessárias para gerenciar fluxos de execução, tomar decisões e repetir tarefas eficientemente.

5.2.1. Estruturas condicionais

As estruturas condicionais são essenciais 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 maneira eficiente. 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.

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 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 da compra  
2 let valorCompra = parseFloat(prompt("Digite o valor da compra: "));  
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 quando há apenas um fluxo alternativo a ser executado. Para situações com duas opções de execução — uma ação se a condição for verdadeira e outra se for falsa — deve-se utilizar a estrutura `if-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` anterior 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 código 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 condição\
5 1 for falsa */
6 } else {
7     /* bloco de código a ser executado se nenhuma das condições anteriores for ve\
8 rdadeira */
9 }
```

Com `else if`, você pode lidar com múltiplas condições sequencialmente, 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, exibe “Bom dia!”. Se a condição não for verdadeira, passa para o próximo `else if`, que verifica se `hora < 18`; se essa condição for verdadeira, exibe “Boa tarde!”. Se nenhuma das condições anteriores for

verdadeira, executa o bloco de código do `else`, que exibe “Boa noite!”. Isso permite que a saudação seja escolhida com base no valor da variável `hora`.

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 são geralmente 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.

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.

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    default:
9      /* código a ser executado se nenhum dos casos anteriores for verdadeiro */
10 }
```

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.

5.2.2. Laços de repetição

Laços de repetição são utilizados para executar um bloco de código várias vezes, com base em uma condição específica. Eles são fundamentais quando se deseja executar tarefas que exigem a 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.

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 ficam 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**: 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**: executado após cada repetição do laço. É onde você modifica a variável usada na condição para controlar a 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, 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 essencial para aprimorar as habilidades 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.

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 laço 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 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, sendo determinada dinamicamente durante a execução do programa. Por exemplo, validar a 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 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.

Laços aninhados

Laços aninhados consistem em colocar um laço de repetição dentro de outro. Eles são úteis para lidar com estruturas de dados bidimensionais, como matrizes, onde o laço externo percorre as linhas e o laço interno percorre 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 de manter.

5.3. 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 e estruturas de controle, você estará preparado para desenvolver programas mais complexos.

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.

5.4. 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 }
```

5.5. Exercícios

1. Implemente um programa que solicita um número ao usuário e seguida exibe uma mensagem informando se o número é par ou ímpar.
2. Escreva um programa que peça ao usuário para inserir três números distintos e, em seguida, imprima o maior deles.
3. Crie um programa que receba 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 conforme a tabela da OMS (abaixo do peso, peso normal, sobrepeso, obesidade). Exiba o valor do IMC e a respectiva classificação.
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 do 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), 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. Em seguida, 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 essenciais na programação, pois agrupam conjuntos de instruções que podem ser reutilizados em diferentes partes do programa, promovendo modularidade e reaproveitamento 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 a criação de códigos mais organizados, legíveis e robustos.

A seguir, abordaremos a declaração, a execução e os tipos de funções, destacando sua importância na programação. Também examinaremos algumas funções predefinidas em JavaScript para fornecer uma compreensão abrangente de seu uso.

6.2. Declaração de funções

A declaração de funções em JavaScript pode ser feita de várias maneiras. Uma das formas mais comuns é usar a palavra-chave `function`, seguida pelo 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 }
```

Esse código define uma função chamada `saudacao` que não recebe parâmetros. Dentro da função, ele exibe a mensagem “Olá, mundo!” no console. Em resumo, a função `saudacao` exibe uma mensagem simples no console quando é chamada.

6.2.1. 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.

Veja o exemplo de uma declaração de função com parâmetro:

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

Esse código define uma função chamada `imprimeQuadradoDe` que recebe um parâmetro `numero`. Dentro da função, ele calcula o quadrado desse número usando o operador `**` (exponenciação) e exibe o resultado no console. Em resumo, a função `imprimeQuadradoDe` calcula o quadrado de um número e o exibe no console.

6.2.2. 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 }
```

Esse código define uma função chamada `soma` que recebe dois parâmetros `a` e `b`. Dentro da função, ele retorna a soma dos dois parâmetros `a` e `b`. A função `soma` retorna a soma dos dois números recebidos como argumentos quando é executada.

6.2.3. 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  };
```

Neste exemplo, a função anônima é criada e atribuída à constante soma para ser invocada posteriormente.

Também é possível declarar funções anônimas usando *arrow function*, que é uma forma concisa e moderna de escrever funções em JavaScript, introduzida no ES6 (ECMAScript 2015). Esse formato oferece 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 são geralmente atribuídas a uma variável ou passadas como argumento para outras funções sem um nome identificador.

Enquanto as funções tradicionais podem ser nomeadas ou anônimas, as *arrow functions* são sempre anônimas.

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 = (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.

6.2.4. Sintaxe de *arrow function*

Arrow function é uma forma mais concisa e moderna de escrever funções em JavaScript. A sintaxe das *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  (parâmetros) => expressão
```

Detalhando a sintaxe da *arrow function*:

- (parâmetros): 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.
- expressão: 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
17 const funcaoComRetornoImplicitoEParenteses = (a, b) => a+b;
18
19 // Com parâmetros e retorno explícito
20 const funcaoComRetornoExplicito = (a, b) => {
21     return a + b;
22 }
23
24 // Com um parâmetro e retorno implícito
25 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. 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.

6.3. 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 ao 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, mundo!" */
7 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     return a + b;
4 };
5
6 /* Chamada da função "soma" com argumentos 2 e 3 que serão atribuídos aos parâmetros
7    a e b respectivamente da função soma.
8    A função soma irá retornar o valor da soma (5), que será armazenado na variável resultado.
9    */
10
11 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 parâmetros e argumentos de funções:

- **Parâmetros:** são variáveis declaradas dentro dos parênteses da função, que esperam receber valores quando a função for chamada.
- **Argumentos:** são os valores reais, passados entre parênteses, na chamada da função.

O uso desses termos como sinônimos é comum na comunidade de programação. No entanto, é importante conhecer as diferenças conceituais entre eles. Abaixo estão alguns pontos importantes sobre a execução de funções:

- Ao chamar uma função, é importante fornecer os valores (argumentos) esperados 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, contribuindo para a modularidade e a manutenibilidade do código.

6.4. A importância do uso de funções

O uso de funções oferece vantagens significativas no desenvolvimento de software:

- **Reutilização:** funções permitem escrever um conjunto de código que pode ser executado várias vezes em diferentes partes do programa, evitando a repetição do mesmo código.
- **Legibilidade:** ao extrair partes do código para funções com nomes descritivos, facilita o entendimento da lógica do programa.
- **Modularidade:** ao dividir o código em funções específicas, o programa se torna mais organizado e fácil de gerenciar, testar e modificar.
- **Manutenibilidade:** funções bem definidas ajudam a melhorar a manutenção do código, permitindo que os desenvolvedores compreendam rapidamente sua funcionalidade e façam correções ou atualizações com mais segurança e eficiência.

Utilize funções para organizar o código, estruturar tarefas complexas e melhorar a legibilidade e a manutenibilidade.

Lembre-se: funções bem nomeadas facilitam a compreensão do seu código.

6.5. 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, fornecendo funcionalidades básicas e comuns, como:

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

Essas funções fornecem funcionalidades 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);
```

O código acima solicita um número ao usuário por meio da função `prompt()`, converte a entrada para um número inteiro usando a função `parseInt()` e, em seguida, exibe o número no console usando a função `console.log()`. Este exemplo ilustra o uso das funções predefinidas para interagir com o usuário, converter tipos de dados e exibir informações no console.

6.6. 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 no 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 de 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 de um bloco de código (como `if`, `else`, `for`, `while`) são locais.
- É o escopo de uma função ou de um 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 variavelGlobal = 'Global';
3
4 function minhaFuncao() {
5     // Variável local de função
6     let variavelLocaFuncao = 'Local de Função';
7
8     /* Acesso permitido à variável global dentro da função */
9     console.log("Variável global:", variavelGlobal);
10
11     /* Acesso permitido à variável local de função dentro da própria função */
12     console.log("Variável local/função:", variavelLocaFuncao);
13
14     // Bloco de código
15     if (true) {
16         // Variável local de bloco de código
17         let variavelDeBloco = 'Local de Bloco';
18
19         // Acesso permitido
20         console.log("Variável global:", variavelGlobal);
21         console.log("Variável local/função:", variavelLocaFuncao);
22         console.log("Variável local/bloco:", variavelDeBloco);
23     }
24
25     /* Acesso a variável local de bloco de código fora do bloco gera erro */
26     console.log("Variável local/bloco:", variavelDeBloco);
27 }
28
29 //Executa função
30 minhaFuncao();
31
32 // Acesso a variáveis globais é permitido em todo arquivo
33 console.log("Variável global:", variavelGlobal);
34
```

```
35  /* Acesso a variável local de função fora da própria função gera erro */  
36  console.log("Variável local/função:", variavelLocalFuncao);
```

Este exemplo demonstra que as variáveis globais são acessíveis em qualquer parte do código, enquanto as variáveis locais de função só podem ser acessadas dentro da função onde foram declaradas. Já as variáveis locais de bloco são limitadas ao bloco em que foram definidas.

Em JavaScript, assim como na maioria das linguagens de programação, o escopo é determinado pelas chaves “{}”. Isso significa que as variáveis declaradas dentro de um bloco de código delimitado por chaves só são acessíveis no 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.

6.7. 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 no código.

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

6.8. 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, calcula 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 variável 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. Usando funções, crie um programa que solicita dois números ao usuário, calcula sua média e exibe o resultado.
11. Use funções para implementar 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 do que o número secreto. Exiba mensagens informando o resultado de cada tentativa e forneça pistas (menor/maior) para auxiliar o usuário.
12. Crie uma calculadora que faça a adição, subtração, multiplicação ou divisão de dois números. Use funções separadas para cada operação e para exibir o resultado.
13. Desenvolva um conversor de moeda que solicita ao usuário o valor em Real e o converte para Dólar. Use funções para ler os dados de entrada, realizar a conversão e exibir o resultado.
14. Usando funções, escreva um programa que solicita ao usuário sua altura e peso e calcula o IMC. Em seguida, exibe o resultado e uma mensagem indicando se a pessoa está abaixo do peso, no peso normal, com sobrepeso ou obesa.
15. Implemente um jogo de adivinhação onde o computador gera um número aleatório entre 1 e 100 e o jogador deve adivinhar qual é o número.
 - O jogador inicia com uma pontuação de 100 pontos.
 - Cada tentativa incorreta reduz 1 ponto da pontuação do jogador.
 - O jogo não possui limite de tentativas, continuando até que o jogador adivinhe corretamente o número secreto.

Quando o jogador acertar, o programa deve:

- Exibir uma mensagem informando que o jogador acertou.
- Informar a pontuação atual do jogador.

Além disso, para auxiliar o jogador:

- Após cada tentativa incorreta, o jogo deve exibir uma dica informando se o número secreto é menor ou maior do que o palpite do jogador.

7. Objetos, Arrays e Strings

No mundo da programação, entender e dominar as estruturas de dados é essencial para desenvolver aplicativos eficientes e robustos.

Neste capítulo, vamos mergulhar em três estruturas de dados fundamentais do JavaScript: objetos, arrays e strings. Essas estruturas são a base de muitas operações e funcionalidades em JavaScript, permitindo-nos organizar, armazenar e manipular informações de maneira eficiente.

Ao explorar cada uma dessas estruturas, vamos entender como elas funcionam, suas características únicas e como aplicá-las em situações reais.

Prepare-se para uma jornada de aprendizado que lhe ajudará a dominar a manipulação de dados em JavaScript. Iniciaremos entendendo o que é uma estrutura de dados e como ela é fundamental para o desenvolvimento de software.

7.1. O que são estruturas de dados?

Estruturas de dados são recursos utilizados para organizar e armazenar dados de maneira eficiente. Elas definem como os dados são armazenados na memória e como podem ser acessados e manipulados pelos 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 a 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. Conhecer essas estruturas é essencial para desenvolver software eficiente e robusto. Neste livro, vamos explorar detalhadamente duas estruturas de dados fundamentais na programação, especialmente para o desenvolvimento de aplicações com JavaScript: objetos e arrays.

7.2. Objetos

Em programação, um objeto é uma estrutura de dados que contém informações e funcionalidades relacionadas. Em JavaScript, um objeto é composto por pares de chave-valor, onde cada chave é um identificador único que representa uma propriedade do objeto, e cada valor associado é o dado correspondente a essa propriedade. Os objetos permitem organizar dados de forma estruturada e representar entidades do mundo real, como pessoas, carros e produtos.

Objetos são uma parte fundamental da programação orientada a objetos e são amplamente utilizados em muitas linguagens de programação, incluindo JavaScript.

Geralmente, objetos são utilizados para representar estruturas de dados complexas, como:

- Criar representações de objetos do mundo real, como usuários, produtos, carros, etc., com suas propriedades e métodos.
- Organizar os dados relacionados em uma única unidade.
- Passar valores como argumentos para funções e métodos.
- Implementar programas orientados a objetos.

Aprender a usar objetos de forma eficaz é fundamental para qualquer desenvolvedor JavaScript. Ao dominar essa poderosa estrutura de dados, você poderá criar aplicações mais organizadas, dinâmicas e escaláveis. A seguir vamos explorar as principais funcionalidades dos objetos para construir programas eficientes.

7.2.1. Criação de objetos

Os objetos podem ser criados de duas maneiras principais: através da sintaxe literal de objeto ou pelo construtor `Object()`.

A sintaxe literal é a forma mais comum de criar objetos. Utilizam-se chaves para definir o objeto e separar os pares chave-valor com vírgulas. Cada par chave-valor é composto por uma chave seguida de dois pontos e do valor. Por exemplo:

```
1 const carro = {  
2   marca: 'Volkswagen',  
3   modelo: 'Gol',  
4   ano: 2020,  
5 };
```

Este código cria um objeto chamado `carro` com três propriedades: `marca`, `modelo` e `ano`, cada propriedade com seu valor correspondente.

O construtor `Object()` permite criar objetos vazios ou inicializá-los com valores específicos, como mostrado abaixo:

```
1 // Cria um objeto vazio
2 const filme = new Object();
3
4 // Cria um objeto com propriedades
5 const carro = new Object({
6   modelo: 'Gol',
7   ano: 2014,
8 });
```

Este código demonstra duas formas de criar objetos em JavaScript utilizando o construtor `Object()`. Na primeira linha, é criado um objeto vazio chamado `filme` por meio do construtor `Object()`. Na segunda linha, é criado um objeto chamado `carro` com duas propriedades: `nome` e `ano`, inicializadas com os valores “Gol” e 2014, respectivamente.

Importante: ao criar um objeto, é essencial ter em mente que esse objeto representa um conjunto de dados que forma um todo coeso. Em outras palavras, as propriedades desse objeto devem estar relacionadas entre si, contribuindo para a representação única desse conjunto de dados.

Por exemplo, ao representar um carro, em vez de ter duas variáveis independentes, como “modeloCarro” e “anoCarro”, é mais apropriado criar um objeto “carro” com propriedades como “modelo” e “ano”. Isso reflete a natureza relacionada dessas informações, onde o modelo e o ano do carro juntos formam uma entidade coesa e bem definida. Portanto, ao criar objetos em JavaScript, é importante considerar a coesão e a relação entre as propriedades, garantindo uma representação adequada e eficiente dos dados.

7.2.2. Manipulação de objetos

Para acessar as propriedades de um objeto, podemos utilizar a notação de ponto ou de colchetes.

A notação de ponto é a forma mais simples e comum de acessar as propriedades. Utiliza-se o nome do objeto seguido de um ponto e o nome da propriedade. Sintaxe: `objeto.propriedade`.

Exemplo:

```
1 const carro = {
2   marca: 'Volkswagen',
3   modelo: 'Gol',
4   ano: 2020,
5 };
6
7 console.log(carro.modelo); // 'Gol'
8 console.log(carro.ano); // 2020
```

Este código acessa as propriedades “modelo” e “ano” do objeto “carro” e imprime seus valores, sendo “Gol” e 2020, respectivamente.

Enquanto a notação de colchetes permite acessar propriedades usando expressões entre colchetes. Essa forma é útil quando o nome da propriedade não é conhecido ou precisa ser gerado dinamicamente. Sintaxe: `objeto[propriedade]`.

Exemplo:

```
1  const carro = {
2    marca: 'Volkswagen',
3    modelo: 'Gol',
4    ano: 2020,
5  };
6
7  const propriedade = 'marca';
8  console.log(carro[propriedade]); // "Volkswagen"
```

O exemplo acima utiliza uma variável `propriedade` para armazenar o nome de uma propriedade do objeto “carro” e acessa essa propriedade dinamicamente através da notação de colchetes.

Para modificar o valor de uma propriedade, basta utilizar a notação de ponto ou de colchetes para acessar a propriedade desejada e atribuir um novo valor a ela. Por exemplo:

```
1  const pessoa = {
2    nome: 'José',
3    idade: 30,
4    profissao: 'Desenvolvedor',
5  };
6
7  // alterando o nome do usuário
8  pessoa.nome = 'Maria';
9  console.log(pessoa);
```

O código acima define um objeto “pessoa” com propriedades como nome, idade e profissão, em seguida, altera o valor da propriedade nome para “Maria” e imprime o objeto atualizado.

Além de acessar e alterar as propriedades de um objeto, podemos adicionar ou remover propriedades e realizar outras operações por meio dos métodos disponíveis em JavaScript. No entanto, esses recursos não serão abordados neste livro, que foca na criação, acesso e modificação de propriedades de objetos.

Os objetos em JavaScript são ferramentas essenciais para organizar, manipular e acessar dados complexos de forma eficiente e intuitiva. Sua versatilidade os torna indispensáveis para o desenvolvimento de aplicações robustas e escaláveis.

Ao criar objetos em JavaScript, sempre busque manter a coerência entre as propriedades. Pense no tema central que o objeto representa e defina as propriedades que realmente estão relacionadas a esse tema. Evite incluir propriedades aleatórias ou sem sentido, pois isso pode tornar seu código confuso e difícil de trabalhar.

Para uma compreensão completa dos objetos em JavaScript, recomendamos consultar a documentação disponível no [MDN Web Docs](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Object)¹.

7.3. Arrays

Array, também conhecido como vetor ou matriz em português, é uma estrutura de dados fundamental na programação, utilizado para armazenar coleções de elementos. Ele pode ser comparado a um armário com gavetas numeradas e ordenadas sequencialmente. Essas gavetas são chamadas de “posições” do array, e cada uma delas é identificada por um número, chamado de **índice**, onde a primeira gaveta recebe o índice 0, a segunda o índice 1, e assim por diante. Esses índices são utilizados para acessar cada elemento do array.

Um array deve ser utilizado quando há a necessidade de armazenar e acessar uma coleção de elementos de forma sequencial e organizada. Isso é útil em situações em que há dados relacionados que precisam ser manipulados juntos, como uma lista de nomes, notas de alunos, entre outros. Principalmente quando a quantidade de elementos é variável.

Em JavaScript, arrays são estruturas de dados que permitem armazenar coleções de elementos, como números, strings, funções, outros arrays, etc. 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 array 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#, 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

¹https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Object

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

7.3.1. Declaração e inicialização

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

Na declaração literal, o array é definido utilizando colchetes [], com os elementos separados por vírgulas. Veja abaixo um exemplo de declaração de um array de frutas com cinco elementos:

```
1 // array com elementos do mesmo tipo: string
2 const frutas = ['Manga', 'Caju', 'Jaca', 'Uva', 'Banana'];
```

A Figura 7.1 ilustra graficamente o array de frutas declarado acima.

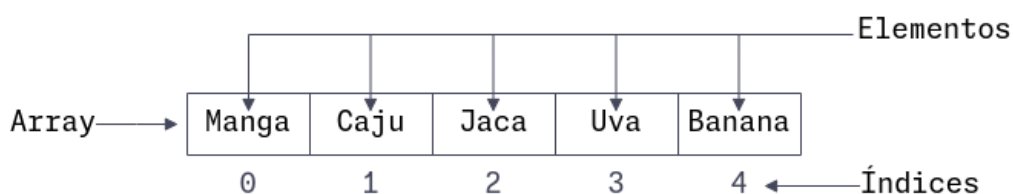


Figura 7.1. array de palavras com cinco elementos

JavaScript permite a criação de arrays homogêneos, com elementos do mesmo tipo, e heterogêneos, que podem conter diferentes tipos de dados, como números, strings, outros arrays, funções, etc. Veja alguns exemplos abaixo:

```
1 // array homogêneo com elementos do mesmo tipo: string
2 const frutas = ['Maçã', 'Banana', 'Laranja'];
3 // array homogêneo com elementos do mesmo tipo: número
4 const numeros = [1, 2, 3, 4, 5];
5 // array heterogêneo com elementos de tipos diferentes
6 const misturado = ['Olá', 10, true];
7 // array vazio com tamanho 0.
8 const carros = [];
```

Outra maneira de criar um array é usando a palavra-chave `new` seguida do construtor `array()`. Desta forma, podemos criar arrays de três maneiras diferentes:

```
1 // Cria um array vazio, sem elementos.
2 const vazio = new array();
3
4 // Cria um array com três elementos.
5 const frutas = new array("Maçã", "Banana", "Laranja");
6
7 // Cria um array com 5 elementos vazios.
8 const numeros = new array(5);
```

Embora o construtor `array` possa ser útil em algumas situações específicas, a declaração literal é geralmente a opção mais flexível e fácil de usar para criar arrays em JavaScript.

7.3.2. Manipulação de arrays

Cada elemento em um array possui um índice que identifica sua posição. Geralmente, é por meio desses índices que acessamos e manipulamos os elementos de um array.

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[indice]` retorna o elemento do índice especificado. 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"
```

Também 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']
```

Além do acesso direto por índice, podemos realizar diversas operações de manipulação em arrays, como adicionar, remover, modificar e pesquisar elementos, utilizando métodos específicos predefinidos. A seguir, vamos descrever e exemplificar os principais métodos e propriedades para manipulação de arrays.

`array.push(elemento)`

Adiciona um ou mais elementos ao final do array.

Exemplo de uso do método `push()` para adicionar um novo elemento a um array.

```
1 let frutas = ['maçã', 'banana'];
2 frutas.push("laranja");
3 console.log(frutas); // ["maçã", "banana", "laranja"]
```

O método `push()` adiciona o elemento especificado (laranja) ao final do array `frutas`.

`array.pop()`

Remove o último elemento do array e o retorna.

Exemplo de como remover o último elemento de um array:

```
1 let frutas = ['maçã', 'banana', 'laranja'];
2 let frutaRemovida = frutas.pop();
3 console.log(frutaRemovida); // "laranja"
4 console.log(frutas); // ["maçã", "banana"]
```

O método `pop()` remove e retorna o último elemento (laranja) do array `frutas`.

`array.shift()`

Remove o primeiro elemento do array e o retorna.

Exemplo de como remover o primeiro elemento de um array:

```
1 let frutas = ['maçã', 'banana', 'laranja'];
2 let frutaRemovida = frutas.shift();
3 console.log(frutaRemovida); // Saída: 'maçã'
4 console.log(frutas); // ["banana", "laranja"]
```

O método `shift()` remove e retorna o primeiro elemento (maçã) do array `frutas`.

`array.unshift(elemento)`

Adiciona um ou mais elementos ao início do array.

Exemplo de uso do método `unshift()` para adicionar um novo elemento no início de um array:

```
1 let frutas = ['banana', 'laranja'];
2 frutas.unshift("uva");
3 console.log(frutas); // ["uva", "banana", "laranja"]
```

O método `unshift()` adiciona o elemento especificado (uva) ao início do array `frutas`.

`array.slice(início, fim)`

Cria um novo array contendo os elementos do array original desde o índice `início` até o índice `fim` (sem incluir o elemento no índice `fim`). Este método não altera o array original, apenas retorna um novo array com os elementos selecionados.

O método `slice()` pode receber dois parâmetros: o primeiro define o índice inicial, que é incluído no novo array. O segundo, opcional, define o índice final, mas o elemento

correspondente a este índice não é incluído. Se o segundo parâmetro não for especificado, todos os elementos a partir do índice inicial até o final do array serão incluídos no novo array.

Exemplo de uso do método `slice()`:

```
1 let frutas = ['maçã', 'limão', 'laranja', 'kiwi'];
2
3 let citricas = frutas.slice(1, 3);
4 console.log(citricas); // ["banana", "laranja"]
5
6 const restoDasFrutas = frutas.slice(2);
7 console.log(restoDasFrutas); // ["laranja", "kiwi"]
```

Este código utiliza o método `slice()` para criar um novo array chamado “citricas” contendo elementos da segunda até a terceira posição (excluindo a posição três) da array original “frutas”, e outro array chamado “restoDasFrutas” com os elementos a partir da segunda posição até o final da array original.

`array.splice(inicio, quantidade)`

O método `splice()` é utilizado para remover elementos de um array, retornando os elementos removidos e modificando o array original.

Para remover elementos de um array com `splice()`, é necessário informar dois parâmetros: o primeiro (índice) indica a posição onde a remoção deve começar, e o segundo (quantidade) especifica o número de elementos a serem removidos a partir do índice inicial.

Exemplo de remoção de elementos de um array:

```
1 let frutas = ['maçã', 'banana', 'laranja', 'uva'];
2 frutas.splice(1, 2); // Remove 2 elementos a partir do índice 1
3 console.log(frutas); // Saída: ["maçã", "uva"]
```

O método `splice()` não apenas remove os elementos especificados, mas também reordena os elementos restantes do array para preencher as lacunas, diminuindo o tamanho do array conforme necessário.

O método `splice()` também pode receber um terceiro parâmetro, que especifica elementos a serem adicionados no lugar dos removidos, permitindo tanto a adição quanto a substituição de elementos em um array. No entanto, seu uso principal é para a remoção de elementos.

`array.concat(outroArray)`

Este método é usado para unir dois ou mais arrays, retornando um novo array que contém todos os elementos dos arrays concatenados. O array original não é modificado.

Exemplo de agrupamento de arrays em um único array.


```
1 let frutas1 = ['maçã', 'banana'];
2 let frutas2 = ['laranja', 'kiwi'];
3 let todasAsFrutas = frutas1.concat(frutas2);
4 console.log(todasAsFrutas); /* ["maçã", "banana", "laranja", "kiwi"] */
```

Este código concatena dois arrays, “frutas1” e “frutas2”, criando um novo array chamado “todasAsFrutas” contendo todos os elementos dos arrays originais.

array.sort()

Ordena os elementos de um array em ordem crescente ou decrescente, seja alfabeticamente ou numericamente.

O método `sort()` não possui parâmetros obrigatórios, no entanto é recomendado fornecer uma função de comparação como parâmetro para personalizar a ordenação. Por padrão, o `sort()` ordena os elementos em ordem crescente.

Exemplo de ordenação numérica de arrays:

```
1 let numeros = [2, 9, 3, 5, 11, 7];
2 numeros.sort((a, b) => a - b);
3 console.log(numeros); // [ 2, 3, 5, 7, 9, 11 ]
```

Esse código ordena os elementos do array `numeros` em ordem crescente através do método `sort()`, que recebe uma função de comparação como argumento. A função de comparação recebe dois parâmetros, `a` e `b`, que representam dois elementos do array. Essa função `(a, b) => a - b` simplesmente subtrai os valores de `a` e `b` e então retorna um valor que indica qual elemento deve vir primeiro na ordem final. Se o resultado for negativo, `a` vem antes de `b`. Se o resultado for positivo, `b` vem antes de `a`. Assim, o `sort()` ordena os números em ordem crescente com base nessa comparação.

Exemplo de ordenação alfabética de arrays:

```
1 let frutas = ['laranja', 'acerola', 'banana', 'açai'];
2 frutas.sort((a, b) => a.localeCompare(b));
3 console.log(frutas); // ['açai', 'acerola', 'banana', 'laranja']
```

Neste exemplo do array `frutas`, o método `sort()` também recebe uma função de comparação que recebe dois parâmetros, `a` e `b`, que representam dois elementos do array. A função então utiliza o método `localeCompare()` para comparar os nomes das frutas. O método `localeCompare()` é um método de comparação sensível à localidade, considera as regras de comparação de strings com caracteres especiais. Isso garante que as strings sejam ordenadas corretamente de acordo com a sua ordem alfabética.

Para ordenar objetos em JavaScript, geralmente precisamos definir uma propriedade de ordenação com base na qual desejamos ordená-los. Isso é necessário porque o método de ordenação precisa de uma base para determinar a ordem dos elementos.

Exemplo de ordenação de um array de objetos:

```
1 // array de objetos
2 const pessoas = [
3   { nome: 'Ana', idade: 30 },
4   { nome: 'João', idade: 25 },
5   { nome: 'Maria', idade: 35 },
6 ];
7
8 /* Ordenar o array de objetos com base na idade, do mais novo ao mais velho */
9 pessoas.sort((a, b) => a.idade - b.idade);
10 // Imprime o array ordenado
11 console.log(pessoas);
```

Esse código cria um array de objetos chamado “pessoas”, onde cada objeto tem duas propriedades: “nome” e “idade”. Em seguida, ele usa o método `sort()` para ordenar o array com base na idade, do mais novo ao mais velho. Isso é feito passando uma função de comparação `((a, b) => a.idade - b.idade)` para o `sort()`, que compara a idade de dois objetos por vez e ordena-os em ordem crescente.

Além da ordenação crescente, também é possível fazer ordenação decrescente utilizando o método `sort()` em conjunto com uma função de comparação que inverte a ordem dos elementos.

Exemplo de ordenação decrescente:

```
1 // Array de strings
2 let frutas = ['laranja', 'acerola', 'banana', 'açai'];
3 frutas.sort((a, b) => b.localeCompare(a));
4 console.log(frutas); // ['laranja', 'banana', 'acerola', 'açai']
5 // Array de números
6 let numeros = [2, 9, 3, 5, 11, 7];
7 numeros.sort((a, b) => b - a);
8 console.log(numeros); // [11, 9, 7, 5, 3, 2]
```

No exemplo acima, foi modificada a ordem dos parâmetros das funções de comparação, colocando `b` antes de `a`, resultando em uma ordenação decrescente nos dois arrays.

Além do método `sort()` para ordenar um array, também podemos inverter a ordem dos elementos utilizando o método `reverse()`. Este método reorganiza os elementos do array na ordem oposta à posição atual deles. Ou seja, se os elementos estão em ordem crescente, após a aplicação do método `reverse()`, eles estarão em ordem decrescente, e vice-versa. Isso é útil quando precisamos modificar a ordem dos elementos de forma rápida e simples.

Exemplo:

```
1 let numeros = [2, 9, 3, 5, 11, 7];
2 // ordem crescente
3 let ordenados = numeros.sort((a, b) => a - b);
4 console.log(ordenados); // [ 2, 3, 5, 7, 9, 11 ]
5 // ordem decrescente
6 ordenados.reverse();
7 console.log(ordenados); // [ 11, 9, 7, 5, 3, 2 ]
```

O método `sort()` permite ordenar os elementos de um array, enquanto `reverse()` inverte a ordem dos elementos. Ambos são úteis para manipular arrays de forma eficiente, seja para ordenação crescente, decrescente ou uma simples inversão da ordem original dos elementos.

array.length

A propriedade `length` retorna um valor numérico indicando a quantidade de elementos presentes no array. Por exemplo:

```
1 let frutas = ['laranja', 'acerola', 'banana', 'açai'];
2 console.log(frutas.length); // 4
```

O exemplo acima usa a propriedade `length` para imprimir o número de elementos no array `frutas`, que é 4.

Iteração com for tradicional

Para percorrer e manipular os elementos de um array, uma abordagem comum é a iteração. Esta técnica permite acessar cada elemento individualmente para realizar operações específicas. Existem várias formas de realizar iterações em JavaScript, sendo o laço `for` uma das mais tradicionais e simples. Por exemplo:

```
1 const frutas = ['maçã', 'banana', 'laranja'];
2
3 for (let i = 0; i < frutas.length; i++) {
4   console.log(frutas[i]); // "maçã", "banana", "laranja"
5 }
```

Neste exemplo, `frutas.length` é usado para determinar o número total de elementos no array `frutas`, garantindo que o laço continue enquanto o índice `i` for menor que o tamanho do array. Dentro do laço, `frutas[i]` é utilizado para acessar cada elemento individualmente, permitindo sua manipulação.

Iteração com for...of

O `for...of` é uma alternativa moderna e simplificada ao tradicional `for` em JavaScript, oferecendo uma maneira mais legível e eficiente de percorrer os elementos de um array. Com o `for...of`, podemos iterar diretamente sobre os valores dos elementos, simplificando a sintaxe e reduzindo a probabilidade de erros. Essa abordagem promove a escrita de laços mais concisos, resultando em um código mais expressivo e fácil de entender.

No `for...of`, a ordem dos elementos é mantida conforme a inserção no array, garantindo consistência na iteração. A sintaxe é simples: `for (const elemento of array) { ... }`.

Exemplo:

```
1 const carros = ['Argo', 'Onix', 'Kwid', 'Gol'];
2
3 for (const elemento of carros) {
4   console.log(elemento); // "Argo", "Onix", "Kwid", "Gol"
5 }
```

Neste código, o laço `for...of` é utilizado para percorrer os elementos do array “carros” e exibir cada elemento no console. Essa construção simplifica a iteração e torna o código mais claro e conciso.

Além da iteração com os laços `for`, JavaScript oferece diversos métodos para iteração e manipulação de arrays, como `forEach()`, `map()`, `filter()` e `find()`. O uso de cada um desses métodos depende da operação a ser realizada. A seguir, vamos explorar os principais métodos de iteração de arrays em JavaScript.

`array.forEach()`

Itera sobre cada elemento do array e executa uma função de retorno de chamada para cada elemento.

Sintaxe: `array.forEach((valorAtual, índice, array) => {...})`, onde `(valorAtual, índice, array) => {...}` é a função (*arrow function*) que será executada para cada elemento do array. Os parâmetros “índice” e “array” são opcionais.

Exemplo de uso do `forEach` para exibir cada elemento de um array.

```
1 const carros = ['Argo', 'Onix', 'Kwid', 'Gol'];
2
3 carros.forEach((carro) =>
4   console.log(carro)
5 );
```

Neste exemplo, o método `forEach()` itera sobre cada elemento do array e imprime o nome de cada carro no console.

`array.map()`

Itera sobre o array original transformando cada elemento e adicionando-o em um novo array.

Sintaxe: `const novoarray = array.map((valorAtual, índice, array) => {...})`, onde `(valorAtual, índice, array) => {...}` é a função (*arrow function*) que será executada para cada elemento do array original e o seu resultado será adicionado como um novo elemento do novoarray. Os parâmetros “índice” e “array” são opcionais.

Exemplo de como aplicar o método `map()` para transformar as letras dos nomes dos carros em maiúsculas

```
1 const carros = ['Argo', 'Onix', 'Kwid', 'Gol'];
2
3 const carrosMaiusculos = carros.map((carro) => carro.toUpperCase());
4
5 console.log(carrosMaiusculos); // ["ARGO", "ONIX", "KWID", "GOL"]
```

Este código usa o método `map()` para converter todos os elementos do array `carros` para letras maiúsculas usando a função `toUpperCase()` e armazena o resultado em `carrosMaiusculos`.

`array.filter()`

Cria um novo array com todos os elementos que atendem a uma condição especificada por uma função.

Sintaxe: `const novoarray = array.filter((valorAtual, índice, array) => {...})` onde `(valorAtual, índice, array) => {...}` é a função (*arrow function*) que aplicará uma condição em cada elemento do array original e caso o elemento atenda a condição será adicionado como um novo elemento do `novoarray`. Os parâmetros “índice” e “array” são opcionais.

Exemplo do uso de `filter()` para encontrar os carros com mais de 3 Letras:

```
1 const carros = ['Argo', 'Onix', 'Kwid', 'Gol'];
2
3 const carrosComMaisDe3Letras = carros.filter((carro) => carro.length > 3);
4
5 console.log(carrosComMaisDe3Letras); // ["Argo", "Onix", "Kwid"]
```

Este código usa o método `filter()` para selecionar os carros no array `carros` que têm mais de 3 letras, baseando-se na condição `carro.length > 3`, e armazena-os em `carrosComMaisDe3Letras`.

`array.includes()`

Verifica se um determinado valor está presente em um array. Ele retorna `true` se o valor for encontrado e `false` caso contrário.

Sintaxe: `array.includes(valor)`.

Exemplo de verificação se determinados nomes de carros estão na array `carros`.

```
1 const carros = ['Argo', 'Onix', 'Kwid', 'Gol'];
2
3 console.log(carros.includes('Onix')); // true
4 console.log(carros.includes('Polo')); // false
```

Este código verifica se o array `carros` inclui os elementos “Onix” e “Polo”, retornando `true` para “Onix” (pois está presente no array) e `false` para “Polo” (pois não está presente no array).

`array.find()`

Retorna o primeiro elemento do array que satisfaz a uma condição especificada por uma função.

Sintaxe: `const elementoEncontrado = array.find((valorAtual, índice, array) => {...})`, onde `(valorAtual, índice, array) => {...}` é a função (*arrow function*) que aplicará uma condição em cada elemento do array original e retornará o primeiro elemento que atenda a condição. Caso nenhum elemento atenda a condição especificada será retornado `undefined`. Os parâmetros “índice” e “array” são opcionais.

Exemplo de como pesquisar um carro específico pelo nome:

```
1 const carros = ['Argo', 'Onix', 'Kwid', 'Gol'];
2
3 const carroEncontrado = carros.find((carro) => carro === 'Gol');
4
5 console.log(carroEncontrado); // 'Gol'
```

Este código utiliza o método `find()` para procurar no array `carros` pelo carro “Gol” e armazená-lo na variável `carroEncontrado`, baseado na condição `carro === 'Gol'`.

Todos os métodos de manipulação de arrays também podem ser aplicados a arrays de objetos. No entanto, ao lidar com objetos, estamos geralmente interessados em operações em propriedades específicas, como buscar, filtrar, modificar ou calcular valores com base em uma propriedade do objeto.

Veja um exemplo de como usar o método `find()` em um array de objetos.

```
1 const pessoas = [
2   { nome: 'Ana', idade: 30 },
3   { nome: 'João', idade: 25 },
4   { nome: 'Maria', idade: 35 },
5 ];
6
7 const pessoaComNomeAna = pessoas.find((pessoa) => pessoa.nome === 'Ana');
8 console.log(pessoaComNomeAna);
```

Neste exemplo, temos um array de objetos chamado `pessoas`, onde cada objeto representa uma pessoa com as propriedades `nome` e `idade`. Usamos o método `find()` para encontrar a primeira pessoa com o nome ‘Ana’ e armazenamos o resultado na variável `pessoaComNomeAna`. Em seguida, imprimimos o objeto encontrado no console.

Além dos métodos mencionados, os arrays em JavaScript oferecem uma ampla gama de funcionalidades úteis para manipulação e iteração de elementos. Entre esses métodos, estão incluídos o `reduce()`, `every()`, `some()`, entre outros.

Para uma visão abrangente desses métodos e suas aplicações, consulte a documentação do JavaScript disponível no [MDN Web Docs](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/array)².

²https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/array

7.4. Strings

String é um tipo de dado que representa uma sequência de caracteres. Em JavaScript, uma string é uma sequência de caracteres entre aspas simples (‘ ’), aspas duplas (“ ”) ou acentos graves (`). As strings podem conter letras, números, espaços, pontuação e caracteres especiais, tornando-se uma ferramenta essencial para manipular texto de forma eficaz.

A linguagem JavaScript oferece algumas maneiras de criar strings. Uma das formas mais comuns é utilizando aspas simples ou duplas, onde os caracteres são inseridos entre elas, como nos exemplos abaixo:

```
1 const textoComAspasSimples = 'Esta é uma string com aspas simples.';
2 const textoComAspasDuplas = "Esta é uma string com aspas duplas.";
```

Outra forma de criar strings é utilizando o acento grave (`) como delimitador, conhecido como *template strings*. Essas strings permitem a interpolação de variáveis e expressões embutidas, facilitando a criação de textos dinâmicos e legíveis. Além disso, elas suportam múltiplas linhas, simplificando a formatação de textos extensos.

No exemplo abaixo, são demonstradas aplicações práticas de template strings:

```
1 const usuario = {
2   nome: 'Maria',
3   idade: 30,
4   profissao: 'Desenvolvedora',
5 };
6
7 // Interpolação de variáveis
8 const mensagem = `Olá, sou ${usuario.nome} e tenho ${usuario.idade} anos.`;
9
10 // Suporte a multiplas linhas
11 const texto = `
12     Este é um exemplo
13     de uma string
14     com multiplas linhas.
15 `;
```

No exemplo acima, a interpolação de variáveis na string “mensagem” permite criar uma mensagem personalizada com base nos dados do objeto “usuário”. Já o suporte a múltiplas linhas na string “texto” facilita escrever textos extensos em múltiplas linhas sem a necessidade de usar caracteres de escape.

Sempre que necessário criar uma string que contenha parte fixa e parte dinâmica, ou se precisar de formatação em múltiplas linhas, é recomendado utilizar template strings. Isso torna o código mais legível, simplifica a concatenação de variáveis e elimina a necessidade de usar caracteres de escape para inserir quebras de linha.

7.4.1. Manipulação de strings

Um aspecto importante das strings é que são imutáveis, ou seja, uma vez criadas, não podem ser alteradas. Porém, JavaScript oferece métodos específicos para manipular strings de forma eficiente.

JavaScript oferece uma variedade de propriedades e métodos para manipular strings, sendo os principais:

string.length

A propriedade `length` retorna a quantidade de caracteres em uma string, incluindo espaços e caracteres especiais.

Exemplo:

```
1 const nome = 'Maria';  
2 console.log(nome.length); // 5
```

string.split()

O método `split()` divide uma string em uma lista ordenada de substrings com base em um separador especificado, colocando essas substrings em um array e retornando-o.

Por exemplo, ao usar o espaço em branco como separador em uma string, o método divide a string em partes sempre que encontra um espaço em branco. O resultado é um array contendo as partes individuais da string original.

Exemplo:

```
1 const nomeCompleto = 'Antonio da Silva';  
2 const nomes = nomeCompleto.split(' ');  
3 console.log(nomes); // ['Antonio', 'da', 'Silva']
```

O exemplo de código acima utiliza o método `split()` para dividir uma string em um array de substrings com base em um delimitador, neste caso, o espaço em branco (“ ”). O resultado é armazenado na variável `nomes`. Portanto, `nomes` se torna um array contendo as palavras individuais “Antonio”, “da” e “Silva”.

Em seguida, o código imprime o array `nomes` no console, resultando em `["Antonio", "da", "Silva"]`, que são as partes obtidas após a divisão da string original.

Conhecer a propriedade `length` e o método `split()` é fundamental para manipular strings de forma eficiente em JavaScript. Com essas duas ferramentas, é possível realizar uma ampla variedade de operações de manipulação de strings, como encontrar substrings, contar o número de ocorrências de determinado caractere, dividir uma string em várias partes e muito mais.

Além da propriedade `length` e do método `split()`, existem outros métodos disponíveis para manipulação de strings em JavaScript, incluindo:

- `charAt(index)`: retorna o caractere na posição especificada.
- `concat(str1, str2, ...)`: concatena duas ou mais strings e retorna uma nova string.
- `indexOf(substring)`: retorna o índice da primeira ocorrência de uma substring na string, ou -1 se não for encontrada.
- `substring(startIndex, endIndex)`: retorna uma parte da string, começando no índice `startIndex` até o índice `endIndex` (não incluso).
- `slice(startIndex, endIndex)`: similar ao método `substring`, mas permite índices negativos para contar a partir do final da string.
- `toUpperCase()`: converte todos os caracteres da string para maiúsculas.
- `toLowerCase()`: converte todos os caracteres da string para minúsculas.
- `trim()`: remove espaços em branco do início e do final da string.

Esses métodos são amplamente utilizados para manipular strings em JavaScript, permitindo realizar diversas operações, como busca, substituição, formatação e validação de texto.

Ao trabalhar com strings, muitas vezes surge a necessidade de percorrer cada caractere individualmente para realizar determinadas operações, como manipulação, busca ou contagem. Para isso, existem várias abordagens, desde os métodos tradicionais até as técnicas mais modernas. Vamos explorar duas maneiras de percorrer eficientemente os caracteres de uma string.

Iteração com `for` tradicional e colchetes

É possível acessar os caracteres individuais de uma string usando a notação de colchetes, assim como se faz com os elementos de um array. E assim, com o uso de um laço `for` podemos iterar sobre cada caractere acessando-o pelo seu índice. Veja o exemplo abaixo:

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

Este exemplo utiliza um laço `for` para imprimir cada caractere da string armazenada na constante `texto`, começando pela primeira posição (0) e indo até a última.

Ao contrário de um array, não é possível modificar diretamente os caracteres individuais de uma string. Se for necessário realizar modificações, é preciso criar uma nova string com as alterações desejadas.

Iteração com `for...of`

O laço `for...of` é uma forma mais moderna e concisa de iterar sobre elementos iteráveis, como strings. Com esse laço, não precisamos nos preocupar com índices ou comprimentos, ele itera diretamente sobre os caracteres da string. Aqui está um exemplo:

```
1 const texto = 'JavaScript';  
2 for (const caractere of texto) {  
3   console.log(caractere);  
4 }
```

Neste caso, cada iteração de `for...of` retorna diretamente o caractere atual da string `texto`, simplificando o processo de iteração e deixando o código mais legível.

Para expandir sua compreensão sobre strings, recomenda-se explorar a documentação do JavaScript no [MDN Web Docs](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/string)³.

7.5. Conclusão

Ao longo deste capítulo, exploramos as estruturas de dados fundamentais em JavaScript — objetos, strings e arrays — fortalecendo nossa compreensão sobre como organizar e manipular informações de forma eficiente.

Os objetos são essenciais para representar dados complexos do mundo real e organizar informações de forma estruturada.

Já os arrays oferecem flexibilidade para armazenar e acessar coleções ordenadas de elementos.

Enquanto as strings são fundamentais para lidar com sequências de caracteres, como texto.

Dominar o uso dessas estruturas é fundamental para o desenvolvimento de aplicações robustas e eficientes em JavaScript.

Parabéns por concluir esta jornada de aprendizado sobre essas estruturas de dados fundamentais em JavaScript! Lembre-se que o aprendizado é um processo contínuo. A seguir você encontrará uma série de exercícios práticos que abrangem desde operações básicas até cenários mais complexos, utilizando as estruturas de dados aprendidas. Vamos lá, hora de colocar a mão no código!

7.6. Exercícios

1. Crie um objeto para representar um aluno, incluindo propriedades como nome, idade, curso e notas. Popule o objeto com valores e imprima os dados do aluno.
2. Crie um objeto carro com as propriedades marca, modelo e ano. Em seguida, imprima a marca do carro.
3. Crie um objeto livro com as propriedades titulo, autor e anoPublicacao. Imprima o título e o autor do livro.
4. Crie um objeto produto com as propriedades nome, preco e quantidade. Em seguida, calcule e imprima o valor total do estoque ($\text{preço} * \text{quantidade}$).

³https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/string

5. Crie uma string com o seu nome e imprima o número de caracteres.
6. Crie uma função que recebe um nome completo do usuário e imprime o primeiro e último nome.
7. Crie uma função que recebe um nome completo do usuário e imprime somente o primeiro nome.
8. Crie uma string com uma frase e use o método `toUpperCase` para imprimir a frase em letras maiúsculas.
9. Escreva um programa que solicita ao usuário o nome de 5 filmes e os armazena em um array. Em seguida, exiba no console os filmes informados.
10. Utilize a lista de filmes do exercício 9 e exiba no console cada um dos filmes informados usando um `for` tradicional.
11. Utilize a lista de filmes do exercício 9 e aplique o método `forEach` para exibir os nomes dos filmes e uma crítica engraçada para cada filme no console.
12. Utilize a lista de filmes do exercício 9 e aplique o método `for...of` para exibir os nomes dos filmes e uma crítica engraçada para cada filme no console.
13. Crie uma nova lista a partir da lista de filmes do exercício 9, utilizando o método `filter`, para incluir apenas dois filmes que você recomendaria aos amigos. Exiba a lista de filmes recomendados no console.
14. Ordene a lista de filmes do exercício 9 por título em ordem alfabética utilizando o método `sort`. Exiba a lista ordenada no console.
15. Implemente uma função que recebe um título de filme como parâmetro e busca na lista de filmes do exercício 9. A função deve retornar uma mensagem informando se o filme foi encontrado ou não na lista. Exiba o resultado da busca no console.
16. Implemente um jogo de adivinhação em que o computador gera um número aleatório entre 1 e 100, e o jogador deve adivinhar qual é o número. O jogo possui as seguintes regras e funcionalidades:
 - a) Pontuação inicial:
 - Cada jogador inicia o jogo com 100 pontos.
 - Cada tentativa incorreta reduz 1 ponto da pontuação do jogador.
 - b) Regras do jogo:
 - O jogo não possui limite de tentativas. O jogador continua jogando até adivinhar corretamente o número secreto.
 - Após cada tentativa incorreta, o programa deve informar se o número secreto é maior ou menor que o palpite do jogador.
 - c) Ao final do jogo:
 - Exibir uma mensagem informando que o jogador acertou o número.

- Exibir a pontuação final do jogador.

d) Registro de jogadores:

- Antes de iniciar o jogo, o jogador deve informar seu nome.
- Ao final do jogo, o nome e a pontuação do jogador devem ser salvos em uma lista.

e) Classificação geral:

- Após o término de cada partida, o programa deve exibir a lista de todos os jogadores e suas respectivas pontuações, ordenada da maior para a menor pontuação.
- Cada entrada na lista deve exibir o nome do jogador e sua pontuação final.

7.6.1. Projeto 1: Lista de compras

Você foi contratado para desenvolver um programa em JavaScript que permita aos usuários gerenciar uma lista de compras. O programa deve oferecer as seguintes funcionalidades:

- **Adicionar um item:** o usuário deve ser capaz de adicionar novos itens à lista de compras;
- **Remover um item:** o usuário deve ser capaz de remover itens da lista de compras;
- **Pesquisar item:** o programa deve permitir que os usuários verifiquem se um determinado item já está na lista de compras.
- **Ordenar a lista:** o programa deve permitir que o usuário ordene a lista de compras pelos nomes dos itens.
- **Exibir lista:** o programa deve permitir a exibição de todos os itens da lista.
- **Limpar a lista:** o programa deve permitir que o usuário remova todos os itens da lista de compras de uma vez.
- **Encerrar programa,** utilize um laço de repetição para manter o programa ativo exibindo o menu de opções até o usuário escolher encerrar o programa.

O programa deve ser interativo, permitindo que o usuário escolha qual funcionalidade deseja executar em cada etapa. Ele deve exibir mensagens informativas para orientar o usuário no processo de gerenciamento da lista de compras.

7.6.2. Projeto 2: Lista de filmes

Crie um programa em JavaScript para gerenciar sua lista de filmes e organizar suas maratonas cinematográficas! O programa deve oferecer as seguintes funcionalidades:

- **Adicionar filme:** os usuários devem poder adicionar novos filmes na lista de filmes para assistir informando o título e o ano de lançamento do filme.

- **Ordenar a lista:** o programa deve permitir ao usuário ordenar a lista de filmes para assistir pelo título ou pelo ano de lançamento para facilitar a visualização.
- **Pesquisar filme:** o programa deve permitir que os usuários verifiquem se um determinado filme já está na lista de filmes para assistir.
- **Exibir listas:** os usuários devem poder exibir tanto a lista de filmes para assistir quanto a lista de filmes já assistidos.
- **Marcar filme como assistido:** Os usuários podem marcar qualquer filme da lista como assistido. Quando um filme é marcado como assistido, ele deve ser removido da lista de filmes para assistir e adicionado à lista de filmes já assistidos.
- **Remover filme:** além de marcar um filme como assistido, os usuários devem ter a opção de remover filmes da lista de filmes para assistir, caso decidam não assisti-los mais.
- **Encerrar programa:** utilize um laço de repetição para manter o programa ativo exibindo o menu de opções até o usuário escolher encerrar o programa.

O programa deve ser interativo, permitindo que o usuário escolha qual funcionalidade deseja executar em cada etapa. Ele deve exibir mensagens informativas para orientar o usuário durante o processo de gerenciamento da lista de compras.

8. Modularização

A modularização (ou componentização) é uma prática essencial no desenvolvimento de software, permitindo organizar e reutilizar o código de forma eficiente. Neste capítulo, exploraremos dois dos principais sistemas de modularização em JavaScript: o CommonJS e os Módulos JavaScript (também conhecidos como Módulos ECMAScript).

Ao longo deste capítulo, examinaremos as características, a sintaxe e os casos de uso de cada um desses sistemas, destacando suas diferenças, vantagens e desvantagens. Compreender essas técnicas de modularização é crucial para o desenvolvimento de aplicativos robustos e escaláveis em JavaScript.

Além disso, também abordaremos o uso de módulos externos, como bibliotecas e pacotes disponibilizados pela comunidade, que desempenham um papel importante na produtividade e expansão das funcionalidades de nossos projetos.

Vamos iniciar entendendo o conceito de modularização na programação.

8.1. Modularização

Em programação, modularização é o processo de dividir um sistema de software em módulos ou unidades independentes, cada uma responsável por uma funcionalidade específica. Essa abordagem facilita o desenvolvimento, a manutenção e a escalabilidade do código, uma vez que permite que diferentes partes do sistema sejam desenvolvidas, testadas e gerenciadas separadamente.

No contexto da programação, o conceito de módulo é bastante flexível e pode variar dependendo do contexto. Um módulo pode ser uma função isolada, um arquivo contendo um conjunto de funções relacionadas, uma parte de um software que agrupa vários arquivos, entre outros.

Neste livro, um módulo é definido como um arquivo que contém uma ou mais funcionalidades relacionadas. Esses arquivos podem exportar (disponibilizar para uso externo) e importar (utilizar de outros módulos) funcionalidades conforme necessário, facilitando a organização e a reutilização de código.

Existem duas maneiras de lidar com a modularização em JavaScript:

- CommonJS: a forma mais antiga, como um padrão para desenvolvimento do lado do servidor, estabelecendo um sistema modular para organizar o código em unidades reutilizáveis chamadas módulos.
- Módulos JavaScript (também conhecido como módulos ECMAScript): introduzidos no ECMAScript 6 (ES6), os módulos JavaScript oferecem uma abordagem mais moderna para a modularização.

A seguir vamos explorar cada uma delas em detalhes. Veremos como criar e usar módulos utilizando o CommonJS e os Módulos JavaScript, destacando suas características, vantagens e cenários de uso adequados.

8.2. CommonJS

CommonJS é a forma mais tradicional de modularização em JavaScript, especialmente em ambientes de servidor como Node.js, que visa organizar e reutilizar código de forma eficiente. Essa abordagem permite dividir o código em arquivos separados, conhecidos como módulos, onde cada um representa uma unidade lógica de funcionalidade.

A sintaxe básica do CommonJS envolve a utilização do `require` para importar módulos e `module.exports` (ou `exports`) para exportar funcionalidades para uso em outros módulos. Por exemplo, o código a seguir demonstra a criação de um módulo utilizando o CommonJS:

```
1 // arquivo operacoes.js
2 const PI = 3.14;
3
4 function soma(a, b){
5     return a + b;
6 }
7
8 function subtracao(a, b) {
9     return a - b;
10 }
11
12 module.exports = { PI, soma, subtracao };
```

Para importar funcionalidades desse módulo (arquivo “operacoes.js”), podemos seguir duas abordagens distintas. Na primeira, conhecida como importação tradicional, todo o conteúdo do módulo é atribuído a uma única variável, como mostrado abaixo:

```
1 // arquivo tradicional.js
2 const operacoes = require('./operacoes');
3
4 // Utilizando as funções e constantes exportadas pelo módulo
5 console.log(operacoes.PI); // Saída: 3.14
6 console.log(operacoes.soma(5, 3)); // Saída: 8
7 console.log(operacoes.subtracao(10, 4)); // Saída: 6
```

Neste código (arquivo “tradicional.js”), estamos importando o módulo “operacoes.js” e atribuindo todas as suas funcionalidades a uma única variável chamada `operacoes`. Isso significa que todas as funções e constantes exportadas pelo módulo “operacoes.js” podem ser acessadas no arquivo “tradicional.js” através da variável `operacoes`, utilizando a notação de ponto para referenciar cada funcionalidade.

Na segunda abordagem, denominada importação individual, cada funcionalidade exportada é importada de forma separada, como ilustrado no exemplo a seguir:

```
1 // arquivo individual.js
2 const { soma, subtracao } = require('./operacoes');
3
4 // Utilizando as funções e constantes exportadas pelo módulo
5
6 console.log(soma(5, 3)); // Saída: 8
7 console.log(subtracao(10, 4)); // Saída: 6
```

Neste exemplo, cada função e constante exportada pelo módulo “operacoes.js” é importada e atribuída a uma variável com o mesmo nome que foi exportada. Isso simplifica o acesso direto às funcionalidades desejadas no arquivo “individual.js”. A importação individual permite importar apenas as funções, variáveis ou constantes específicas que serão utilizadas no módulo, tornando o código mais conciso e facilitando o acesso aos elementos desejados.

Em geral, recomendo a importação individual, por tornar o código mais organizado, legível e fácil de manter.

No entanto, ambas as formas de importação são válidas e eficazes, e a escolha entre elas pode depender das preferências do desenvolvedor ou do estilo de codificação adotado pela equipe.

8.3. Módulos JavaScript

Os Módulos JavaScript oferecem uma abordagem moderna e flexível para organizar e reutilizar código em aplicações. Permitem a divisão do código em arquivos separados, facilitando a manutenção e promovendo uma estrutura mais modular e organizada para o desenvolvimento de software.

Em Módulos JavaScript, há dois tipos de exportação: padrão e nomeada.

8.3.1. Exportação padrão

A exportação padrão permite exportar um único valor por módulo, geralmente uma função ou objeto principal. Veja um exemplo:

```
1 // moduloMatematicaPadrao.js
2 function somar(a, b) {
3   return a + b;
4 }
5
6 // exportando a função "somar" como exportação padrão
7 export default somar;
```

No exemplo acima, utilizamos a expressão `export default` para exportar a função `somar` como uma exportação padrão do arquivo “moduloMatematicaPadrao.js”.

Ao importar esse módulo em outro arquivo, podemos atribuir qualquer nome desejado a ele, como mostrado abaixo:


```
1 // moduloPrincipalPadrao.js
2 import somar from './moduloMatematicaPadrao';
3
4 const resultado = somar(5, 3);
5 console.log(resultado); // Exibe: 8
```

Neste exemplo, importamos o módulo do arquivo “moduloMatematicaPadrao.js” e atribuímos o nome `soma` para ele. O nome atribuído deve ser coerente com a funcionalidade importada.

8.3.2. Exportação nomeada

As exportações nomeadas permitem exportar múltiplos valores ou objetos com nomes específicos. Por exemplo:

```
1 // moduloMatematicaNomeado.js
2 export function somar(a, b) {
3   return a + b;
4 }
5
6 export function subtrair(a, b) {
7   return a - b;
8 }
9
10 export function multiplicar(a, b) {
11   return a * b;
12 }
```

Neste exemplo, exportamos individualmente cada função utilizando a palavra-chave `export`.

Ao importar esses itens, devemos usar os mesmos nomes especificados na exportação, como no exemplo a seguir:

```
1 // moduloPrincipalNomeado.js
2 import { somar, multiplicar } from './moduloMatematicaNomeado';
3
4 const resultadoSoma = somar(10, 20);
5 const resultadoMultiplicacao = multiplicar(3, 5);
6
7 console.log(resultadoSoma); // 30
8 console.log(resultadoMultiplicacao); // 15
```

Esse exemplo importa individualmente as funções `somar` e `multiplicar` do módulo “moduloMatematicaNomeado.js” usando a mesma nomenclatura e as utiliza para realizar operações matemáticas.

Diferença entre exportação padrão e nomeada:

- Exportação padrão
 - Permite exportar apenas um item por módulo.
 - Permite a atribuição de qualquer nome na importação.
- Exportação nomeadas
 - Permite exportar vários itens com nomes específicos.
 - Exige que os mesmos nomes sejam utilizados na importação.

A escolha entre exportação padrão e exportações nomeadas depende das necessidades específicas do projeto ou do estilo de codificação adotado pela equipe de desenvolvimento.

8.4. Módulos externos

Módulos externos, também conhecidos como bibliotecas ou pacotes, são arquivos de código JavaScript que oferecem funcionalidades prontas para uso em seus projetos. Eles encapsulam código reutilizável, simplificando o desenvolvimento e evitando a repetição desnecessária de código.

O uso de módulos externos em JavaScript é uma prática comum para ampliar as funcionalidades da linguagem, permitindo que os desenvolvedores aproveitem bibliotecas criadas por terceiros para resolver problemas específicos ou adicionar recursos extras aos seus projetos.

O [NPM](https://www.npmjs.com/)¹ (Node Package Manager) é um repositório e gerenciador de pacotes para JavaScript. Ele facilita a instalação, atualização e remoção de módulos externos de maneira eficiente. Ao usar o NPM, você tem acesso a um vasto repositório online com milhares de módulos para diversas tarefas, desde manipulação de dados até a criação de interfaces gráficas. O NPM é instalado automaticamente com o Node.js, portanto, para utilizá-lo, é necessário instalar o Node.js em sua máquina.

Para utilizar um módulo externo em um projeto JavaScript, é necessário instalá-lo previamente. Em projetos Bun, isso pode ser feito usando o comando `bun install nome-biblioteca` ou podemos usar o próprio NPM com o comando `npm install nome-biblioteca`. Essa instrução instala a biblioteca no projeto e, além disso, cria ou atualiza o arquivo `package.json`. Este arquivo é o local de configuração central de um projeto JavaScript e é onde são listadas as dependências do projeto, incluindo os módulos externos necessários.

O Bun também tem a funcionalidade de gerenciar dependências, podendo ser utilizado no lugar do NPM.

Ao executar o comando `npm install nome-biblioteca` ou `bun add nome-biblioteca`, o NPM ou Bun baixará a versão mais recente da biblioteca especificada do repositório NPM e a instalará localmente no projeto. O arquivo `package.json` será atualizado automaticamente

¹<https://www.npmjs.com/>

para incluir uma entrada para a biblioteca “mathjs” na seção de dependências. Isso permite que o projeto gerencie facilmente as dependências e garanta que outros desenvolvedores repliquem o ambiente de desenvolvimento com facilidade.

Um exemplo prático do uso de módulos externos em JavaScript é a biblioteca “mathjs”, que fornece um conjunto abrangente de funções matemáticas para realizar operações complexas.

Para instalar a biblioteca “mathjs” em seu projeto, abra um terminal e digite o seguinte comando:

```
1 bun add mathjs
```

Após a instalação, você pode importar a biblioteca “mathjs” em seus arquivos JavaScript como um Módulo JavaScript e utilizar as funções dela em seu código. Por exemplo, vamos usar a função `sqrt` da biblioteca “mathjs” para calcular a raiz quadrada de um número:

```
1 // Importando a função sqrt do módulo mathjs
2 const { sqrt } = require('mathjs');
3
4 // Calculando a raiz quadrada de 25
5 const resultado = sqrt(25);
6
7 // Exibindo o resultado
8 console.log(resultado); // 5
```

Este exemplo importa a função `sqrt` do módulo “mathjs”, usada para calcular a raiz quadrada de 25, e exibe o resultado.

Para explorar as funcionalidades e recursos oferecidos por uma biblioteca externa, é essencial consultar sua documentação correspondente. Por exemplo, ao verificar a [documentação](#)² de “mathjs”, você pode encontrar a função `sqrt` disponibilizada para calcular a raiz quadrada de um número.

Os módulos externos são ferramentas essenciais para o desenvolvimento de aplicações em JavaScript. Ao utilizar bibliotecas consolidadas pela comunidade, você expande as capacidades do JavaScript, torna seu código mais eficiente e aumenta sua produtividade.

Explore o vasto repositório de módulos do NPM e descubra como eles podem ajudá-lo a construir projetos web incríveis! A escolha dos módulos deve ser feita com base nas necessidades específicas do seu projeto.

8.5. Conclusão

A modularização é uma prática fundamental no desenvolvimento de software, pois nos permite organizar e reutilizar o código de forma eficiente, facilitando a manutenção, o compartilhamento e a escalabilidade dos projetos.

²<https://www.npmjs.com/package/mathjs>

Ao comparar a modularização com CommonJS e Módulos JavaScript (ES Modules), podemos observar que ambos têm suas vantagens e casos de uso específicos.

CommonJS é amplamente utilizado em ambientes Node.js e oferece uma abordagem simples e funcional para modularizar o código. No entanto, Módulos JavaScript (ES Modules) são mais modernos, possuem uma sintaxe mais clara e flexível, e são suportados em uma variedade de ambientes, incluindo navegadores, Node.js, Bun e TypeScript.

Portanto, para novos projetos, recomendo o uso de Módulos JavaScript.

Ao incorporar módulos externos, como bibliotecas e pacotes disponíveis no NPM, os desenvolvedores podem ampliar ainda mais as funcionalidades de seus projetos, aproveitando o código criado pela comunidade e reduzindo o tempo de desenvolvimento.

8.6. Exercícios

1. Pesquise e identifique três vantagens de usar a modularização em projetos de software. Em seguida, escreva um parágrafo explicando cada vantagem e forneça exemplos de como ela pode ser aplicada em um projeto real.
2. Pesquise sobre o uso do CommonJS e Módulos JavaScript nos últimos 5 anos.
3. Usando CommonJS, crie um módulo que contém funções matemáticas básicas (soma, subtração, multiplicação, divisão) e exponenciação. Em seguida, utilize essas funções em outro arquivo.
4. Usando Módulos JavaScript, crie um módulo que contém funções matemáticas básicas (soma, subtração, multiplicação, divisão) e exponenciação. Em seguida, utilize essas funções em outro arquivo.
5. Crie um módulo que representa um banco de dados de usuários, armazenando informações como nome, e-mail e senha de usuários. Utilize uma lista para armazenar os dados privados do usuário e crie funções para adicionar, remover e atualizar informações dos usuários. Somente as funções devem ser visíveis para módulos externos.
6. Escreva um programa em JavaScript que utiliza a biblioteca [mathjs](https://www.npmjs.com/package/mathjs)³ para calcular o logaritmo de um número fornecido pelo usuário.
7. Escreva um programa em JavaScript que utiliza a biblioteca [convert](https://www.npmjs.com/package/convert)⁴ para converter dias em minutos e gigabytes em bytes.

³<https://www.npmjs.com/package/mathjs>

⁴<https://www.npmjs.com/package/convert>

9. Programação Orientada a Objetos com JavaScript

Um paradigma de programação é uma abordagem ou estilo de escrever código que determina a estrutura, organização e forma como os programas são desenvolvidos. Existem diversos paradigmas de programação, cada um com suas próprias regras, conceitos e técnicas. Entre eles, a Programação Orientada a Objetos (POO) se destaca como um dos mais utilizados no mundo da programação.

Neste capítulo, nosso objetivo é oferecer uma introdução aos conceitos básicos de POO em JavaScript. Não pretendemos nos aprofundar excessivamente, mas sim fornecer uma base sólida para a compreensão e a aplicação desses conceitos fundamentais.

9.1. Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) é um paradigma de programação que organiza o código em torno de “objetos”. Esses objetos são entidades que representam elementos do mundo real, cada um com suas características e comportamentos específicos.

Na POO, as características dos objetos são chamadas de **atributos** ou **propriedades**. Já os comportamentos, as ações que o objeto pode realizar, são chamados de **métodos**.

Por exemplo, no desenvolvimento de um aplicativo de streaming de música como Spotify e Deezer, podemos criar objetos que representam músicas, artistas, playlists, entre outros. Um objeto “música” pode conter atributos como título, duração e compositor, com métodos para iniciar e pausar a reprodução.

Para compreender a POO, é essencial familiarizar-se com alguns conceitos básicos, como classe, objeto, atributo e método.

- **Classe:** uma classe é um modelo ou estrutura que define as características e comportamentos de um objeto. Ela serve como um tipo de molde a partir do qual os objetos são criados. Por exemplo, podemos ter uma classe chamada “Carro”, que define as características e ações comuns a todos os carros, como modelo, cor e métodos para acelerar e frear.
- **Objeto:** um objeto é uma instância específica de uma classe. Em outras palavras, é uma realização concreta do modelo definido pela classe. Utilizando o exemplo anterior, um objeto específico da classe “Carro” pode ser um Gol Vermelho.
- **Atributo:** um atributo é uma característica de um objeto que o descreve. Em termos de programação, os atributos são as variáveis associadas a um objeto. No contexto do objeto “carro”, os atributos podem incluir cor, modelo, ano de fabricação, velocidade, entre outros.

- **Método:** um método é uma função associada a um objeto que define seu comportamento ou ações que o objeto pode realizar. Métodos são utilizados para manipular os atributos de um objeto ou para realizar operações específicas. No exemplo do objeto “carro”, podemos ter métodos como “acelerar”, “frear” e “ligar”. Esses métodos alteram o estado do objeto (por exemplo, aumentando a velocidade ao acelerar).

Esses conceitos básicos são fundamentais para entender POO e são amplamente utilizados na construção de sistemas de software modernos.

9.2. POO com JavaScript

JavaScript suporta POO nativamente, permitindo aos desenvolvedores criar sistemas complexos e escaláveis de forma organizada e eficiente.

Para demonstrar na prática, vamos criar a estrutura de um aplicativo de Rede Social usando os fundamentos da POO em JavaScript. Nesse exemplo, utilizaremos classes para modelar Usuários, Publicações e Comentários, cada uma com suas próprias características e funcionalidades. Assim, podemos entender como a POO auxilia na organização, modularidade e reutilização do código.

O primeiro passo para o desenvolvimento de um programa usando o POO é definir as classes, atributos e métodos para em seguida implementar o código. Dessa forma, o nosso aplicativo de Rede Social será composto por três classes principais: `Usuario`, `Publicacao` e `Comentario`, cada classe conterá os atributos e métodos relacionados conforme descritos abaixo:

Classe `Usuario`: representa um usuário da rede social.

- **Atributos:**
 - `nome`: nome do usuário.
 - `dataNascimento`: data de nascimento do usuário.
 - `seguidores`: lista de usuários (objetos da própria classe `Usuario`) que seguem o usuário.
 - `seguindo`: lista de usuários (objetos da própria classe `Usuario`) que o usuário segue.
- **Métodos:**
 - `criarPublicacao(texto, imagens, videos)`: Cria uma nova publicação no perfil do usuário.
 - `seguirUsuario(usuario)`: começa a seguir outro usuário.
 - `deixarDeSeguirUsuario(usuario)`: deixa de seguir outro usuário.

Classe `Publicacao`: representa uma publicação na rede social.

- **Atributos:**

- `usuario`: objeto da classe `Usuario` que representa o autor da publicação.
 - `dataPublicacao`: data e hora da publicação.
 - `descricao`: conteúdo textual da publicação.
 - `midia`: imagem ou vídeo da publicação (opcional).
 - `curtidas`: número de curtidas na publicação.
 - `comentarios`: lista de comentários na publicação.
- Métodos:
 - `editar(novoTexto, novaMidia)`: edita o conteúdo da publicação.
 - `remover()`: remove a publicação do sistema.
 - `curtir()`: incrementa o contador de curtidas da publicação.
 - `descurtir()`: decrementa o contador de curtidas da publicação.
 - `comentar(comentario)`: adiciona um novo comentário à publicação.

Classe `Comentario`: representa um comentário em uma publicação.

- Atributos:
 - `usuario`: objeto da classe `Usuario` que representa o autor do comentário.
 - `dataComentario`: data e hora do comentário.
 - `texto`: Conteúdo textual do comentário.
- Métodos:
 - `editar(novoTexto)`: edita o conteúdo do comentário.
 - `remover()`: remove o comentário do sistema.

Agora que definimos as classes, seus atributos e métodos, chegou a hora de transformá-los em código. Utilizando JavaScript e os conceitos de POO, vamos implementar essas classes.

Classe Usuário (*Usuario.js*):

```
1 import Publicacao from "../Publicacao.js";
2
3 class Usuario {
4   constructor(nome, dataNascimento) {
5     this.nome = nome;
6     this.dataNascimento = dataNascimento;
7     this.seguidores = [];
8     this.seguindo = [];
9     this.publicacoes = [];
10  }
11
12  /* Este método possibilita a criação de uma nova publicação, onde são fornecido\
```

```
13  s o texto e possíveis mídias (como imagens ou vídeos), e então uma nova instância
14  da classe Publicacao é criada com esses dados, associada ao usuário que a criou,
15  e adicionada à sua lista de publicações. */
16  criarPublicacao(texto, midia) {
17      const novaPublicacao = new Publicacao(this, texto, midia);
18      this.publicacoes.push(novaPublicacao);
19  }
20
21  /* Este método permite que o usuário atual comece a seguir outro usuário, adici\
22  onando-o à sua lista de "seguindo" e também se registrando na lista de "seguidore
23  s" daquele usuário. */
24  seguirUsuario(usuario) {
25      this.seguindo.push(usuario);
26      usuario.seguidores.push(this);
27  }
28
29  deixarDeSeguirUsuario(usuario) {
30      // Método para deixar de seguir outro usuário
31  }
32 }
33
34 export default Usuario;
```

A classe `Usuario` representa um usuário na aplicação de rede social. Ela possui atributos como nome, data de nascimento, seguidores e “seguindo”. Além disso, possui métodos para criar publicações, seguir e deixar de seguir outros usuários.

Vamos aproveitar este primeiro exemplo de classe para entender a sua estrutura. Uma classe em JavaScript é uma estrutura que nos permite criar objetos com propriedades e métodos associados. No exemplo fornecido, temos a classe `Usuario`. Aqui está uma explicação dos principais elementos dessa classe.

A classe é definida usando a palavra-chave `class`, seguida pelo nome da classe (`Usuario`, neste caso). As chaves delimitam o escopo da classe, ou seja, marcam seu início e fim.

Os atributos de uma classe são variáveis que armazenam os dados de cada instância da classe. Eles são declarados dentro do construtor.

O construtor, definido por meio da palavra-chave *constructor*, é um método especial chamado automaticamente quando um objeto é instanciado a partir da classe. Ele é responsável por inicializar os atributos do objeto.

No exemplo acima, o construtor da classe `Usuario` recebe três argumentos: `nome` e `dataNascimento`. Dentro do construtor, a palavra-chave `this` se refere ao próprio objeto que está sendo criado. Por exemplo, `this.nome` se refere ao atributo `nome` do objeto atual. Quando um objeto é criado a partir da classe `Usuario`, os argumentos passados para o construtor são usados para inicializar os atributos do objeto. Por exemplo, se criarmos um novo usuário com `const usuario1 = new Usuario("José", "25-03-1940");`, os valores “José” e “25-03-1940” serão atribuídos aos atributos `nome`, e `dataNascimento`, respectivamente.

No construtor, também é possível definir valores padrão para os atributos da classe. Por exemplo, os atributos seguidores e seguindo são inicializados como arrays vazios. Isso garante que todos os objetos da classe `Usuario` são criados com esses valores para esses atributos.

Os métodos são funções definidas dentro da classe que podem ser chamados em instâncias dessa classe. Eles são declarados sem a necessidade de usar a palavra-chave `function`. Eles descrevem o comportamento dos objetos criados a partir da classe. No exemplo, temos três métodos: `criarPublicacao`, `seguirUsuario` e `deixarDeSeguirUsuario`, que realizam ações específicas relacionadas aos usuários.

Por fim, a instrução `export default` é usada para exportar a classe `Usuario`, permitindo que ela seja importada e utilizada em outros arquivos JavaScript. Isso significa que podemos importar essa classe em outro arquivo usando a instrução `import`.

A estrutura básica da classe, portanto, inclui a definição da classe usando `class`, seguida pelo nome da classe, o construtor para inicializar os atributos da classe, métodos para definir o comportamento dos objetos e, opcionalmente, a exportação da classe para uso em outros arquivos.

Embora o construtor não seja obrigatório, é uma boa prática defini-lo para ter mais controle sobre a criação e inicialização dos objetos da classe.

- Classe Publicação (*Publicacao.js*):

```
1 import Comentario from "../Comentario.js";
2
3 class Publicacao {
4   constructor(usuario, descricao, midia) {
5     this.usuario = usuario;
6     this.dataPublicacao = new Date(); // Pega a data atual
7     this.descricao = descricao;
8     this.midia = midia;
9     this.curtidas = 0;
10    this.comentarios = [];
11  }
12
13  editar(novoTexto, novaMidia) {
14    // Método para editar o conteúdo da publicação
15  }
16
17  remover() {
18    // Método para remover a publicação do sistema
19  }
20
21  /* Este método é responsável por incrementar o contador de curtidas da publica\
22  ão. Quando um usuário curte uma publicação, esse método é chamado, e o contador d
23  e curtidas é aumentado em 1. */
24  curtir() {
```

```
25     this.curtidas++;
26   }
27
28   descurtir() {
29     // Método para decrementar o contador de curtidas da publicação
30   }
31
32   /* Este método permite adicionar um novo comentário à publicação. Quando um usu\
33   ário comenta em uma publicação, este método é invocado. Ele cria uma nova instânc
34   ia da classe Comentario, utilizando o usuário que está comentando e o texto do co
35   mentário. Em seguida, esse novo comentário é adicionado ao array comentarios da p
36   ublicação. */
37   comentar(texto) {
38     const novoComentario = new Comentario(this.usuario, texto);
39     this.comentarios.push(novoComentario);
40   }
41 }
42
43 export default Publicacao;
```

A classe `Publicacao` representa uma publicação feita por um usuário na aplicação. Ela contém atributos como o usuário que fez a publicação, a data e hora da publicação, uma descrição, mídia (imagem ou vídeo), número de curtidas e uma lista de comentários. Possui métodos para editar, remover, curtir, descurtir e comentar uma publicação.

- Classe Comentário (*Comentario.js*):

```
1 class Comentario {
2   constructor(usuario, texto) {
3     this.usuario = usuario;
4     this.dataComentario = new Date();
5     this.texto = texto;
6   }
7
8   editar(novoTexto) {
9     // Método para editar o conteúdo do comentário
10  }
11
12  remover() {
13    // Método para remover o comentário do sistema
14  }
15 }
16
17 export default Comentario;
```

A classe `Comentario` representa um comentário feito por um usuário em uma publicação. Ela possui atributos como o usuário que fez o comentário, a data e hora do comentário e o texto do comentário. Contém métodos para editar e remover o comentário.

Agora que já temos nossas estruturas de classes prontas (Usuario, Postagem e Comentario), vamos criar os objetos, que são as entidades concretas da aplicação. São esses objetos que armazenam os dados, executam ações e interagem entre si durante a execução do sistema real.

Arquivo *main.js*

```
1 // Importando as classes
2 import Usuario from './Usuario.js';
3
4 // Criando dois objetos de usuário
5 const usuario1 = new Usuario("Alice", new Date("1990-05-15"));
6 const usuario2 = new Usuario("Bob", new Date("1985-10-22"));
7
8 // Fazendo com que o usuário 2 siga o usuário 1
9 usuario2.seguirUsuario(usuario1);
10
11 // Criando uma publicação para o usuário 1
12 usuario1.criarPublicacao("Primeira publicação somente texto", null);
13
14 // Curtindo a própria publicação
15 usuario1.publicacoes[0].curtir();
16
17 // Adicionando comentário à própria publicação
18 usuario1.publicacoes[0].comentar("Isto é um comentário");
19
20 // Usuário curtindo publicação de outro usuário
21 usuario2.seguindo[0].publicacoes[0].curtir();
22
23 // Exibindo nomes dos usuários
24 console.log(usuario1.nome);
25 console.log(usuario2.nome);
26 // Exibindo quantidade de seguidores dos usuários
27 console.log(usuario1.seguidores.length);
28 console.log(usuario2.seguidores.length);
29 // Exibindo quantidade de "seguindo" dos usuários
30 console.log(usuario1.seguindo.length);
31 console.log(usuario2.seguindo.length);
32
33 // Exibindo das publicações do Usuário 1
34 console.log(usuario1.publicacoes);
35 console.log(usuario1.publicacoes[0].curtidas);
36 console.log(usuario1.publicacoes[0].comentarios);
```

Esse código simula interações entre dois usuários em uma rede social. Vamos analisar cada parte:

- Linha 2: a classe `Usuario` é importada do arquivo `Usuario.js`.

- Linhas 5 e 6: criação de usuários, dois objetos de usuário são criados, `usuario1` e `usuario2`, com nomes 'Alice' e 'Bob', respectivamente, e datas de nascimento.
- Linha 9: o usuário 2 (`usuario2`) começa a seguir o usuário 1 (`usuario1`) por meio do método `seguirUsuario`.
- Linha 12: o usuário 1 (`usuario1`) cria uma publicação com o texto 'Primeira publicação somente texto' utilizando o método `criarPublicacao`.
- Linha 15: o usuário 1 (`usuario1`) curte sua própria publicação acessando a primeira publicação em seu array `publicacoes` e chamando o método `curtir`.
- Linha 18: o usuário 1 (`usuario1`) adiciona um comentário à sua própria publicação acessando a primeira publicação em seu array `publicacoes` e chamando o método `comentar`.
- Linha 21: o usuário 2 (`usuario2`) curte a publicação do usuário 1 (`usuario1`) acessando a primeira publicação no array `publicacoes` do usuário que o usuário 2 está seguindo.
- Linha 24 em diante: usa-se o `console.log()` para exibir informações dos usuários e publicações.

No geral, o código demonstra como os métodos da classe `Usuario` e da classe `Publicacao` são usados para realizar interações básicas em uma rede social, como criar publicações, curtir e comentar em publicações.

9.3. Conclusão

Neste capítulo, apresentamos uma introdução à Programação Orientada a Objetos (POO) em JavaScript.

Exploramos os conceitos fundamentais de classes, objetos, atributos e métodos, e como esses elementos são utilizados na modelagem de sistemas complexos, como uma aplicação de rede social.

No entanto, é importante ressaltar que a POO é um campo vasto e complexo, com princípios e conceitos avançados, como abstração, encapsulamento, herança e polimorfismo, que não foram abordados aqui.

Se você estiver interessado em se aprofundar em POO e dominar seus princípios mais avançados, recomendamos buscar conteúdos específicos e dedicados a esse tema, como livros completos e recursos online especializados.

A compreensão desses conceitos mais avançados é essencial para se tornar um programador mais habilidoso e eficiente na construção de sistemas de software robustos e escaláveis.

9.4. Exercícios

1. Complete a implementação do aplicativo de Rede Social, adicionando implementação para todos os métodos de todas as classes.

2. Sistema de gerenciamento de biblioteca: imagine que você foi contratado para desenvolver um sistema de gerenciamento de biblioteca em JavaScript utilizando POO. O sistema deve permitir que os usuários realizem operações como adicionar livros, emprestar livros, devolver livros, entre outras funcionalidades comuns em uma biblioteca. Aqui estão algumas das entidades (classes) do sistema: Livro, Usuário, Empréstimo, Biblioteca, etc.
3. Implemente um aplicativo de músicas semelhante ao Spotify e Deezer. Aqui estão alguns requisitos básicos para o sistema:

Classe Música:

- Atributos: titulo, artista, duracao, arquivo(arquivo mp3).
- Métodos: reproduzir() e pausar().

Classe Artista:

- Atributos: nome, foto, musicas.
- Métodos: listaDeMusicas(), adicionarMusica(musica), removerMusica(musica).

Classe Playlist:

- Atributos: nome, musicas.
- Métodos: adicionarMusica(musica), removerMusica(musica), reproduzir(), avancarMusica(), voltarMusica().

10. Testes Automatizados

No desenvolvimento de software atual, os testes automatizados são essenciais. Eles garantem que o código seja robusto, confiável e atenda aos requisitos. Neste capítulo, vamos explorar a importância desses testes, suas vantagens e como aplicá-los efetivamente em projetos de software.

Durante o capítulo, vamos aprender a escrever testes automatizados em JavaScript, uma habilidade crucial para qualquer desenvolvedor. Abordaremos desde os conceitos básicos até os diferentes tipos de testes e ferramentas de automação. Por fim, veremos exemplos práticos para ilustrar como aplicar esses conhecimentos em seu trabalho.

Para reforçar o aprendizado, incluiremos uma série de exercícios práticos. Com essas atividades, você poderá aprimorar suas habilidades em testes automatizados. Ao final do capítulo, você estará mais preparado para criar e manter sistemas de software de alta qualidade, com confiança e eficiência. Vamos começar!

10.1. Introdução

Teste de software refere-se ao processo de verificar se um determinado sistema, aplicativo ou módulo funciona conforme o esperado. Há duas maneiras de realizar esses testes: manualmente ou de forma automatizada.

No teste manual, o usuário interage diretamente com o código, inserindo dados e verificando manualmente se os resultados estão corretos. Nesse método, o usuário desempenha o papel de executor do teste, realizando ações e observando os resultados manualmente.

Já no teste automatizado, o processo é conduzido por programas ou *scripts*. Os testes são escritos em código para serem executados automaticamente. Isso envolve a criação de cenários de teste que definem as entradas esperadas e a execução desses testes por meio de um programa automatizado. Esse programa simula a execução do código original, inserindo as entradas especificadas e verificando automaticamente os resultados.

Enquanto o teste manual depende da interação humana, o teste automatizado utiliza programas e ferramentas para automatizar o processo, proporcionando eficiência, confiabilidade e escalabilidade aos testes de software.

Os testes automatizados devem ser executados regularmente durante o desenvolvimento de software, geralmente como parte de uma estratégia de integração contínua. Isso garante que as alterações recentes no código não tenham introduzido problemas no sistema.

Os testes automatizados oferecem várias vantagens em relação aos testes manuais, como a repetição consistente e a execução rápida em grande escala. Além disso, permitem que os testes sejam realizados automaticamente sempre que há alterações no código-fonte, garantindo a detecção precoce de regressões ou problemas.

Em suma, teste automatizado é uma ferramenta indispensável para os desenvolvedores que buscam construir software de alta qualidade e eficiência. Ele é fundamental para garantir que o código esteja funcionando corretamente e para permitir que os desenvolvedores trabalhem de maneira ágil e segura.

Agora, vamos explorar os diferentes tipos de testes automatizados e examinar exemplos práticos de automação de testes utilizando JavaScript.

10.2. Tipos de testes automatizados

Dentro do ecossistema dos testes automatizados, existem vários tipos, cada um com sua própria finalidade e escopo. Os principais tipos são:

- **Testes unitários** verificam o funcionamento de unidades individuais de código, geralmente funções ou métodos, isolando-as de suas dependências externas. Os testes unitários são rápidos de executar e dão feedback imediato sobre a integridade das unidades de código testadas.
- **Testes de integração** avaliam a interação entre diferentes componentes ou módulos do sistema. Eles visam garantir que essas partes se integrem corretamente e funcionem em conjunto conforme o esperado.
- **Testes de ponta a ponta** simulam o fluxo de interações do usuário com o sistema, do início ao fim. Esses testes são projetados para validar a funcionalidade do software em um contexto que se aproxima ao uso real.

Além desses tipos principais, existe uma variedade de outros tipos de testes automatizados, incluindo testes de desempenho, testes de segurança, testes de usabilidade e muito mais. Cada um desses tipos tem sua própria importância e contribui para a garantia da qualidade do software em diferentes aspectos.

10.3. Ferramentas para automação de testes em JavaScript

Existem várias ferramentas e frameworks disponíveis para automatizar testes em JavaScript, cada uma com suas próprias vantagens. A escolha depende das necessidades e preferências individuais, assim como do tipo de teste desejado.

O *Jest* e o *Cypress* estão entre as ferramentas mais populares. O *Jest* é um framework abrangente usado para testes unitários e de integração, tanto no servidor, com Node.js, quanto no cliente, em aplicações web. Sua flexibilidade e facilidade de uso o tornam muito adotado pelos desenvolvedores.

Por outro lado, o *Cypress* é focado em testes de ponta a ponta, especialmente para automação de testes de interface de usuário. Ele permite simular interações reais do usuário em aplicações web, validando o comportamento do aplicativo de forma abrangente.

No entanto, o Bun e o Node.js (a partir da versão 20) oferecem recursos nativos para automatizar testes em aplicações JavaScript. Isso elimina a necessidade de uso de ferramentas externas. Essa integração nativa simplifica o processo de desenvolvimento e facilita a adoção de práticas de teste automatizado em projetos JavaScript, tornando-o mais eficiente e acessível para os desenvolvedores.

10.4. Testes automatizados na prática

Neste livro introdutório à programação, concentramo-nos em testes unitários como uma forma de demonstrações práticas de automação. Ao explorar os testes unitários, os leitores terão a oportunidade de aprender conceitos essenciais de teste gradualmente, acompanhando exemplos práticos e exercícios. Isso proporcionará uma base sólida para a compreensão de outros tipos de testes automatizados no futuro, à medida que os leitores ganham confiança e habilidade em suas capacidades de programação e teste.

Como já utilizamos o Bun como ambiente de execução ao longo do livro, optamos por utilizar o “bun test” como ferramenta de testes automatizados, simplificando a configuração inicial. O “bun test” oferece uma abordagem direta e eficiente para escrever e executar testes, proporcionando uma experiência simplificada aos desenvolvedores. Além disso, é importante ressaltar que o “bun test” é compatível com o *Jest*, permitindo uma transição suave entre essas duas ferramentas e concedendo maior flexibilidade aos desenvolvedores para escolher a que melhor se adequa às suas necessidades e preferências.

No primeiro exemplo deste capítulo, vamos criar testes automatizados para as funções contidas no módulo “operacoes-modulo.js”, criado anteriormente. Abaixo está o código funcional do módulo:

```
1 // operacoes-modulo.js
2 export function somar(a, b) {
3   return a + b;
4 }
5
6 export function subtrair(a, b) {
7   return a - b;
8 }
9
10 export function multiplicar(a, b) {
11   return a * b;
12 }
```

Antes de iniciar os testes automatizados, é crucial definir os cenários de testes, que são situações específicas que descrevem o comportamento esperado do sistema em diferentes circunstâncias. Cada cenário inclui condições de entrada, ações e resultados esperados para validar o funcionamento correto do software.

Em nosso exemplo inicial, vamos estabelecer dois cenários distintos para cada função do módulo ‘operacoes-modulo.js’: um com dois números positivos e outro com um número

positivo e um negativo como argumentos. Veja abaixo os testes automatizados para esses cenários:

```
1 //arquivo operacoes-modulo.test.js
2
3 // Importando as funções describe, expect, test do módulo bun:test
4 import { describe, expect, test } from "bun:test";
5 // Importando as funções a serem testadas
6 import { somar, subtrair, multiplicar } from '../capitulo-8/operacoes-modulo';
7
8 // Testes da função somar()
9 describe('Testes para a função somar', () => {
10   test('Dois números positivos', () => {
11     expect(somar(2, 3)).toBe(5);
12   });
13
14   test('Um número positivo e um negativo', () => {
15     expect(somar(5, -3)).toBe(2);
16   });
17 });
18
19 // Testes da função subtrair()
20 describe('Testes para a função subtrair', () => {
21   test('Dois números positivos', () => {
22     expect(subtrair(5, 2)).toBe(3);
23   });
24
25   test('Um número positivo e um negativo', () => {
26     expect(subtrair(5, -3)).toBe(8);
27   });
28 });
29
30 // Testes da função multiplicar()
31 describe('Testes para a função multiplicar', () => {
32   test('Dois números positivos', () => {
33     expect(multiplicar(2, 3)).toBe(6);
34   });
35
36   test('Um número positivo e um negativo', () => {
37     expect(multiplicar(5, -3)).toBe(-15);
38   });
39 });
```

O objetivo desses testes é garantir que as funções somar, subtrair e multiplicar estejam se comportando corretamente em diferentes situações, incluindo números positivos e negativos. Vamos explorar o código de testes acima.

Na primeira linha de código deste exemplo (linha 4), importamos as funções `describe`, `expect` e `test` do módulo “bun:test”. Agora vamos entender o propósito de cada uma dessas funções:

- **describe:** função utilizada para agrupar testes relacionados, permitindo uma organização mais clara e estruturada do código de teste. Ela recebe dois argumentos: uma string que descreve o conjunto de testes e uma função de *callback* que contém os testes a serem executados dentro desse grupo.
- **test:** esta função é usada para definir um caso de teste específico dentro de um bloco *describe*. Ela recebe dois argumentos: uma string que descreve o caso de teste e uma função de *callback* que contém o teste a ser executado.
- **expect:** função utilizada para fazer afirmações sobre o comportamento do código que está sendo testado. Ela recebe um valor como argumento e retorna um objeto que possui métodos para verificar se o valor atende às expectativas definidas pelo teste.
- **toBe:** é uma das funções de comparação mais comuns utilizadas em testes automatizados. Ela é usada para verificar se o valor esperado é exatamente igual ao valor retornado pela função sendo testada.

Ao utilizarmos `expect(algumaExpressao).toBe(valorEsperado)`, a ferramenta de testes irá avaliar se `algumaExpressao` (o valor retornado pela função em teste) é exatamente igual a `valorEsperado`. Se a condição for atendida, o teste será bem-sucedido; caso contrário, o teste falhará.

Para executar o código de testes, basta utilizar o comando `'bun test nome-arquivo'`. Por exemplo, para executar os testes no arquivo `'operacoes-modulo.test.js'`, basta digitar `'bun test operacoes-modulo.test.js'` no terminal.

É importante destacar que os arquivos de testes devem obrigatoriamente ter `“.test”` no nome do arquivo para que o comando `“bun test”` consiga identificar e executar esses arquivos como testes automatizados.

Após a execução dos testes, o Bun gera um relatório de testes que informa a quantidade de testes bem-sucedidos e os testes que falharam. Em caso de falha, o relatório também apresenta onde ocorreu, fornecendo informações detalhadas para facilitar a identificação e correção do problema. Esses relatórios são essenciais para identificar e corrigir eventuais falhas de forma rápida e eficiente.

Agora vamos abordar os testes automatizados para a classe “Usuario” desenvolvida no capítulo anterior. Abaixo está o código funcional da classe “Usuario” (arquivo *Usuario.js*):

```
1 import Publicacao from './Publicacao.js';
2
3 class Usuario {
4   constructor(nome, dataNascimento) {
5     this.nome = nome;
6     this.dataNascimento = dataNascimento;
7     this.seguidores = [];
8     this.seguindo = [];
9     this.publicacoes = [];
10  }
11 }
```

```
12  /* Este método possibilita a criação de uma nova publicação, onde são fornecidos\  
13  o texto e possíveis mídias (como imagens ou vídeos), e então uma nova instância  
14  da classe Publicacao é criada com esses dados, associada ao usuário que a criou,  
15  e adicionada à sua lista de publicações. */  
16  criarPublicacao(texto, midia) {  
17      const novaPublicacao = new Publicacao(this, texto, midia);  
18      this.publicacoes.push(novaPublicacao);  
19  }  
20  
21  /* Este método permite que o usuário atual comece a seguir outro usuário, adici\  
22  onando-o à sua lista de "seguindo" e também se registrando na lista de "seguidore  
23  s" daquele usuário. */  
24  seguirUsuario(usuario) {  
25      this.seguindo.push(usuario);  
26      usuario.seguidores.push(this);  
27  }  
28  
29  deixarDeSeguirUsuario(usuario) {  
30      // Método para deixar de seguir outro usuário  
31  }  
32 }  
33  
34 export default Usuario;
```

Para a classe `Usuario`, vamos estabelecer dois cenários de testes distintos. O primeiro cenário será destinado ao método `criarPublicacao`, a fim de verificar se as publicações estão sendo criadas corretamente na classe. Já o segundo cenário será para o método `seguirUsuario`, onde verificaremos se a ação de seguir um usuário está sendo registrada adequadamente. Veja o código abaixo:

```
1  // arquivo Usuario.test.js  
2  import { describe, expect, test, beforeEach } from "bun:test";  
3  import Usuario from '../capitulo-9/Usuario.js';  
4  
5  describe('Testes para a classe Usuario', () => {  
6      let usuario1;  
7      let usuario2;  
8  
9      beforeEach(() => {  
10         usuario1 = new Usuario('Manoel', '15-05-1935');  
11         usuario2 = new Usuario('Francisca', '10-10-1940');  
12     });  
13  
14     test('Teste para criarPublicacao', () => {  
15         // Teste com publicações vazias  
16         expect(usuario1.publicacoes.length).toBe(0);  
17  
18         usuario1.criarPublicacao('Olá mundo!');
```

```
19     usuario1.criarPublicacao('Estou testando a criação de publicações.');
```

```
20
```

```
21     // Verifica se o número de publicações foi incrementado corretamente
```

```
22     expect(usuario1.publicacoes.length).toBe(2);
```

```
23   });
```

```
24
```

```
25   test('Teste para seguirUsuario', () => {
```

```
26     // Teste com seguidores e seguindo vazios
```

```
27     expect(usuario1.seguidores.length).toBe(0);
```

```
28     expect(usuario1.seguindo.length).toBe(0);
```

```
29
```

```
30     usuario1.seguirUsuario(usuario2);
```

```
31
```

```
32     // Verifica se o usuário foi adicionado à lista de seguindo
```

```
33     expect(usuario1.seguindo).toContain(usuario2);
```

```
34
```

```
35     // Verifica se o usuário atual foi adicionado à lista de seguidores do outro \
```

```
36     usuário
```

```
37     expect(usuario2.seguidores).toContain(usuario1);
```

```
38   });
```

```
39
```

```
40 };
```

Vamos explicar o código acima, destacando cada parte e os cenários de testes implementados.

1. **import:** importamos as funções `describe`, `expect`, `test` e `beforeEach` do módulo `bun:test` e a classe `Usuario` a ser testada.
2. **describe:** utilizamos `describe` para agrupar os testes relacionados à classe `Usuario`.
3. **beforeEach:** utilizamos o `beforeEach` para criar duas variáveis `usuario1` e `usuario2` como usuários para serem usados nos testes. O `beforeEach` é um método de ciclo de vida fornecido pela ferramenta de testes que é executado antes de cada teste definido pela função `test()` no bloco `describe`. Neste caso, estamos utilizando-o para garantir que `usuario1` e `usuario2` sejam redefinidos antes de cada teste, garantindo que cada teste comece com os objetos em seu estado inicial.
4. **usuario1 e usuario2:** são instâncias da classe `Usuario` que são criadas antes de cada teste, utilizando os dados fornecidos. Esses objetos são usados para realizar operações e verificar resultados nos testes.
5. **test:** utilizamos a função `test` para descrever e criar cada cenário de teste.
6. Cenário de teste para `criarPublicacao`: este teste verifica o comportamento da função `criarPublicacao`. Ele começa verificando se a lista de publicações do `usuario1` está vazia. Em seguida, chama a função `criarPublicacao` duas vezes para adicionar duas publicações. Por fim, verifica se o número de publicações do `usuario1` foi incrementado corretamente.
7. Cenário de teste para `seguirUsuario`: este teste verifica o comportamento da função `seguirUsuario`. Primeiro, verifica se as listas de seguidores e seguindo do `usuario1` estão

vazias. Depois, chama a função `seguirUsuario` para fazer com que o `usuario1` siga o `usuario2`. Em seguida, verifica se o `usuario2` foi adicionado corretamente à lista de seguindo do `usuario1` e se o `usuario1` foi adicionado corretamente à lista de seguidores do `usuario2`.

Observamos que os testes acima desempenham efetivamente seu objetivo de garantir que as funções `criarPublicacao` e `seguirUsuario` estejam se comportando conforme o esperado.

Dessa forma, recomenda-se que o desenvolvimento de testes automatizados seja realizado em paralelo com o código funcional. Essa prática não apenas facilita a execução de testes repetitivos, mas também proporciona maior produtividade e confiança durante as etapas de desenvolvimento. Com os testes automatizados, possibilita-se identificar rapidamente quaisquer erros ou problemas no código funcional, permitindo diagnósticos ágeis e precisos. Assim, garante-se a integridade e estabilidade do software, mesmo diante de alterações ou modificações, contribuindo significativamente para a qualidade do produto final.

É fundamental escrever testes que cubram o máximo de cenários possíveis, ou seja, testar diferentes entradas e condições para cada funcionalidade. Essa abordagem é crucial para assegurar que o código seja robusto e confiável em diversas situações. Ao testar uma variedade de entradas para cada funcionalidade, podemos identificar e corrigir potenciais problemas antes que eles afetem a aplicação em produção. Adicionalmente, testar diferentes cenários nos permite validar a funcionalidade em diversos contextos, garantindo que o software atenda às expectativas do usuário em todas as situações possíveis.

Em resumo, escrever testes abrangentes é uma prática essencial para garantir a qualidade e a estabilidade do software ao longo do tempo.

10.5. Conclusão

Os testes automatizados representam um pilar fundamental no desenvolvimento de software moderno. Compreendidos como conjuntos de procedimentos que verificam se um sistema funciona corretamente, esses testes oferecem uma série de vantagens essenciais para os desenvolvedores. Além de garantir a integridade e a estabilidade do código, eles facilitam a identificação precoce de falhas, promovendo uma abordagem proativa na resolução de problemas.

Em JavaScript, o desenvolvimento de testes automatizados é particularmente acessível. Ferramentas como `Node.js`, `Bun`, *Jest* e *Cypress*, oferecem estruturas robustas para a criação e execução de testes, tornando o processo ágil e eficiente. Por meio dessas ferramentas, os desenvolvedores podem escrever e executar testes de forma rápida e precisa, garantindo a qualidade e a confiabilidade de seus produtos.

É crucial destacar que o domínio dos testes automatizados é um conhecimento essencial para desenvolvedores. Além de agregar valor ao currículo profissional, a habilidade de criar e implementar testes automatizados demonstra um compromisso com a excelência e a qualidade do código desenvolvido.

Recomendo também que o leitor explore outros tipos de testes automatizados, principalmente os testes de integração, que considero como os testes que mais agregam valor ao desenvolvimento de software. Esses testes avaliam a interação entre diferentes partes do sistema, identificando potenciais problemas de integração que podem não ser detectados nos testes unitários. Ao combiná-los com os testes unitários, os desenvolvedores podem garantir uma cobertura abrangente do código e aumentar a confiabilidade de todo o sistema.

Em suma, investir em testes automatizados é investir na qualidade e no sucesso de um projeto de software. Ao adotar essa prática, os desenvolvedores podem reduzir significativamente o tempo gasto na identificação e correção de falhas, ao mesmo tempo, em que aumentam a confiabilidade e a satisfação do usuário final. Portanto, é importante que os desenvolvedores busquem aprimorar suas habilidades nesse campo, visando alcançar padrões mais elevados de excelência técnica e profissional.

10.6. Exercícios

1. Adicione mais cenários de teste para as funções de operações matemáticas do primeiro exemplo, incluindo operações com strings em vez de números. Se necessário, faça alterações no código funcional para que se adeque melhor aos novos cenários.
2. Crie testes automatizados para todos os métodos da classe `Usuario`.
3. Crie testes automatizados para todos os métodos da classe `Publicacao` (desenvolvida no capítulo anterior).
4. Crie testes automatizados para todos os métodos da classe `Comentario` (desenvolvida no capítulo anterior).
5. Escreva testes automatizados para os exercícios dos capítulos anteriores.
6. Pesquise sobre testes de integração (ferramentas e boas práticas) em aplicações Node.js.

11. Introdução ao TypeScript

TypeScript é uma linguagem de programação de código aberto que estende o JavaScript, adicionando recursos de tipagem estática opcional e outros elementos da Programação Orientada a Objetos (POO). Em essência, o TypeScript proporciona uma série de vantagens que incluem detecção de erros de tipo durante o desenvolvimento, suporte a conceitos de POO, compatibilidade total com JavaScript e ferramentas de desenvolvimento robustas. Este capítulo oferece uma visão geral das principais características e diferenças entre TypeScript e JavaScript.

11.1. O que é TypeScript?

TypeScript é uma linguagem de programação de código aberto desenvolvida pela Microsoft. Ela é uma extensão do JavaScript que adiciona recursos de tipagem estática opcional e outros recursos de programação orientada a objetos. Em resumo, o TypeScript oferece:

- **Tipagem estática opcional:** permite aos desenvolvedores especificar tipos de dados para variáveis, parâmetros de função e retornos de função. Isso ajuda a detectar erros de tipo em tempo de desenvolvimento, reduzindo a ocorrência de erros em tempo de execução.
- **Recursos de programação orientada a objetos:** TypeScript suporta conceitos familiares de programação orientada a objetos, como classes, interfaces, herança, polimorfismo e encapsulamento, tornando-o mais adequado para desenvolvimento de grande escala e colaborativo.
- **Compatibilidade com JavaScript:** qualquer código JavaScript válido é automaticamente considerado válido em TypeScript, possibilitando aos desenvolvedores adotarem gradualmente o TypeScript em seus projetos existentes.
- **Compilação para JavaScript:** o código TypeScript é compilado para JavaScript antes de ser executado em um navegador ou em qualquer outro ambiente JavaScript. Isso significa que os navegadores e ambientes JavaScript existentes podem executar código TypeScript sem a necessidade de suporte nativo.
- **Ambiente de desenvolvimento:** a maioria dos editores de texto e *IDEs* têm suporte robusto para TypeScript, oferecendo recursos como autocompletar, refatoração de código e análise estática, o que pode melhorar significativamente a produtividade e a qualidade do código.

O código TypeScript (extensão `.ts`) precisa ser compilado para JavaScript (extensão `.js`) antes da execução, utilizando o compilador TypeScript, conhecido como `tsc`, que converte arquivos TypeScript em JavaScript. Por exemplo, para compilar um arquivo `index.ts`, basta executar o

comando `tsc index.ts`, o que criará um arquivo `index.js` com o código compilado, em seguida o arquivo `index.js` pode ser executado. Esse processo é necessário tanto em ambientes de servidor quanto em navegadores.

No entanto, com o Bun, o código TypeScript pode ser executado diretamente da mesma forma que JavaScript, pois o Bun possui suporte embutido para TypeScript. Isso significa que não é necessário compilar o código antes da execução. Por exemplo, para executar um arquivo `index.ts`, você pode usar o comando `bun index.ts`, que compila e executa automaticamente o código.

A sintaxe do TypeScript é bastante similar à do JavaScript, a principal distinção está na capacidade de realizar declarações estáticas dos tipos de dados e no suporte aos recursos avançados da Programação Orientada a Objetos (POO). A seguir, vamos explorar as principais diferenças entre TypeScript e JavaScript.

11.2. Declaração de variáveis

Em TypeScript, os tipos de dados podem ser explicitamente declarados para variáveis e constantes. Para declarar uma variável com um tipo específico em TypeScript, você usa a sintaxe `let nomeDaVariavel: tipo;` onde `nomeDaVariavel` é o nome da variável e `tipo` é o tipo de dados que a variável irá armazenar.

```
1 let numero: number;
```

Neste exemplo, declara-se uma variável chamada `numero` e explicita que ela só pode armazenar valores numéricos (tipo `number`). Essa sintaxe é útil quando se deseja garantir que uma variável seja do tipo esperado, proporcionando maior clareza e segurança ao código. Dessa forma, a variável `numero` só aceita valores numéricos, como `numero = 10;`. Se tentarmos atribuir um valor de outro tipo, como `numero = '5';`, o compilador do TypeScript irá acusar um erro e o código não será compilado. Neste caso, a própria ferramenta de edição de código indicará que há um erro.

Quando declaramos e inicializamos uma variável na mesma linha, o TypeScript utiliza inferência de tipos para determinar o tipo automaticamente, garantindo que apenas valores do mesmo tipo da inicialização sejam aceitos, tornando o código menos verboso. Por isso, é comum omitir o tipo na declaração de constantes, já que elas são sempre inicializadas no momento da declaração. Por exemplo:

```
1 const nome = 'José';
```

Neste exemplo, não é necessário nem recomendado adicionar explicitamente o tipo de dado para a constante `nome`, pois o TypeScript infere que ela é do tipo `string` com base no valor atribuído, `'José'`. Adicionar o tipo explicitamente, torna o código redundante e mais verboso. Portanto, nesse caso é uma prática mais limpa e recomendável deixar o TypeScript inferir o tipo com base no valor atribuído.

Vale destacar que quando o tipo não é explicitamente informado, o TypeScript utiliza a inferência de tipos para identificar automaticamente o tipo das variáveis, parâmetros e retorno de funções, facilitando o desenvolvimento ao tornar o código mais conciso e legível.

11.3. Declaração de funções

Em TypeScript, as funções podem ter seus tipos de parâmetros e de retorno declarados explicitamente. Para especificar o tipo dos parâmetros, utiliza-se a mesma sintaxe usada para declarar o tipo de variáveis. Da mesma forma, para definir o tipo de retorno de uma função, utiliza-se a sintaxe `: tipo` entre os parênteses e as chaves da definição da função. Por exemplo, se quisermos criar uma função que recebe dois números como parâmetros e retorna sua soma, podemos fazer o seguinte:

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

Neste exemplo, `a: number` e `b: number` declaram os parâmetros da função `soma` como números, e `: number` após os parênteses indica que a função retorna um valor do tipo número. Essa explicitação dos tipos de parâmetros e retorno não apenas torna o código mais legível, mas também ajuda o TypeScript a detectar erros de tipo durante o desenvolvimento.

11.4. Interfaces

Além da tipagem estática, o TypeScript oferece outros recursos que vão além do JavaScript. Um exemplo são as interfaces. Uma interface em TypeScript é uma forma de definir a estrutura de um objeto, especificando quais propriedades e métodos ele deve conter.

As interfaces são utilizadas para definir tipos, garantindo que os objetos que as implementam possuam determinadas propriedades e métodos, o que ajuda a assegurar a consistência e a integridade do código.

No capítulo sobre objetos, criamos exemplos de objetos para representar carros, cada um com uma quantidade diferente de atributos. Usando interfaces do TypeScript, podemos padronizar e definir uma estrutura obrigatória para todos os objetos carro.

Por exemplo, podemos definir uma interface `Carro` que especifica quais propriedades devem estar presentes em qualquer objeto que a implemente:

```
1 interface Carro {  
2     marca: string;  
3     modelo: string;  
4     ano: number;  
5     cor: string;  
6     calcularIdade(): number; /* Método para calcular a idade do carro */  
7 }
```

Neste exemplo, a interface Carro define quatro propriedades: marca (string), modelo (string), ano (number) e cor (string). Ela também possui um método chamado calcularIdade que não recebe parâmetros e retorna um valor do tipo number.

Com isso, garantimos que qualquer objeto do tipo Carro que criarmos deve ter as propriedades e métodos especificados na interface Carro, mantendo a consistência e a integridade do código. Se tentarmos criar um objeto carro sem uma das propriedades obrigatórias, o TypeScript nos alertará durante a compilação, ajudando a prevenir erros.

Agora, podemos criar objetos do tipo Carro que seguem a estrutura na Interface:

```
1 /* Criação do objeto carro1 que segue a estrutura definida na interface Carro */  
2 let carro1: Carro = {  
3     marca: 'Toyota',  
4     modelo: 'Corolla',  
5     ano: 2020,  
6     cor: 'Prata',  
7     calcularIdade() {  
8         return new Date().getFullYear() - this.ano;  
9     }  
10 };  
11  
12 /* Criação do objeto carro2 que segue a estrutura definida na interface Carro */  
13 let carro2: Carro = {  
14     marca: 'Honda',  
15     modelo: 'Civic',  
16     ano: 2018,  
17     cor: 'Preto',  
18     calcularIdade() {  
19         return new Date().getFullYear() - this.ano;  
20     }  
21 };  
22  
23 /* Definição da função imprimirDadosCarro que recebe um parâmetro do tipo Carro e\  
24 não retorna nada (void) */  
25 function imprimirDadosCarro(carro: Carro): void {  
26     console.log("Marca:", carro.marca);  
27     console.log("Modelo:", carro.modelo);  
28     console.log("Ano:", carro.ano);  
29     console.log("Cor:", carro.cor);  
30     console.log("Idade:", carro.calcularIdade());
```

```
31 }  
32  
33 /* Chamada da função imprimirDadosCarro passando o objeto carro1 como argumento */  
34 imprimirDadosCarro(carro1);  
35  
36 imprimirDadosCarro(carro2);
```

Neste código, os objetos `carro1` e `carro2` são do tipo `Carro`, o que implica que devem incluir todas as propriedades e métodos definidos na interface `Carro`.

A função `imprimirDadosCarro` recebe um parâmetro `carro` do tipo `Carro`, o que garante que ela possa exibir cada atributo do objeto `carro` com segurança, ao ter a garantia de que o objeto terá todos os atributos definidos na interface `Carro`. Isso ajuda a garantir a consistência e a integridade do código, evitando erros relacionados a atributos ausentes ou métodos inconsistentes nos objetos do tipo `Carro`.

As interfaces em TypeScript permitem definir contratos para objetos, garantindo que eles tenham propriedades e métodos específicos. Esses contratos ajudam a manter a consistência no código, facilitam a colaboração em equipe e previnem erros de implementação. Ao utilizar interfaces, tornamos o código mais legível e autodocumentado, promovendo a criação de projetos robustos e fáceis de manter.

11.5. Classes e modificadores de acesso

Em TypeScript, assim como em JavaScript, as classes são utilizadas para definir estruturas de dados e comportamentos seguindo o paradigma orientado a objetos. No entanto, o TypeScript apresenta um recurso adicional: os modificadores de acesso. Esses modificadores controlam a visibilidade de atributos e métodos dentro das classes. Essa abordagem segue os princípios fundamentais da POO, permitindo o encapsulamento e restringindo o acesso a partes específicas do código. Os modificadores de acesso mais comuns são `public`, `private` e `protected`, onde:

- `public`: os membros são acessíveis de qualquer lugar.
- `private`: os membros são acessíveis apenas de dentro da própria classe.
- `protected`: os membros são acessíveis dentro da própria classe e de suas subclasses.

Observe o exemplo abaixo, onde a classe `Usuario` do aplicativo de rede social desenvolvido no Capítulo 9 foi reescrita em TypeScript.

```
1  import Publicacao from "../Publicacao";
2
3  class Usuario {
4      private nome: string;
5      private dataNascimento: Date;
6      private seguidores: Usuario[] = [];
7      private seguindo: Usuario[] = [];
8      private publicacoes: Publicacao[] = [];
9
10     public constructor(nome: string, dataNascimento: Date) {
11         this.nome = nome;
12         this.dataNascimento = dataNascimento;
13     }
14
15     getNome(){
16         return this.nome;
17     }
18
19     getQtdSeguidores(){
20         return this.seguidores.length;
21     }
22
23     getQtdSeguindo(){
24         return this.seguindo.length;
25     }
26
27     getSeguindo(){
28         return this.seguindo;
29     }
30
31     getSeguidores(){
32         return this.seguidores;
33     }
34
35     getPublicacoes(){
36         return this.publicacoes;
37     }
38
39     public criarPublicacao(texto: string, midia: string) {
40         const novaPublicacao = new Publicacao(this, texto, midia);
41         this.publicacoes.push(novaPublicacao);
42     }
43
44     public seguirUsuario(usuario: Usuario) {
45         this.seguindo.push(usuario);
46         usuario.seguidores.push(this);
47     }
48
49     public deixarDeSeguirUsuario(usuario: Usuario) {
```

```
50      // Método para deixar de seguir outro usuário
51    }
52  }
53
54  export default Usuario;
```

Os modificadores de acesso na classe `Usuario` controlam a visibilidade dos membros da classe, como propriedades e métodos. Aqui está o uso de cada modificador de acesso:

- `private`: os membros marcados como `private` são acessíveis somente dentro da própria classe. Neste exemplo, as propriedades `nome`, `dataNascimento`, `seguidores`, `seguindo` e `publicacoes` são marcadas como `private`, o que significa que elas só podem ser acessadas e modificadas dentro dos métodos da classe `Usuario`.
- `public`: os itens marcados como `public` são acessíveis de fora da classe. Todos os métodos na classe `Usuario` são marcados como `public`, o que significa que eles podem ser chamados e utilizados de fora da classe por outros objetos.

Esses modificadores de acesso ajudam a encapsular o comportamento da classe, restringindo o acesso direto às suas propriedades internas e permitindo que a interação com a classe seja feita via métodos públicos, promovendo um código mais seguro e modular.

Quando um atributo ou método não tem um modificador de acesso explícito, ele recebe o modificador padrão que é `public`. Isso significa que ele pode ser acessado de fora da classe, da mesma forma que se fosse explicitamente declarado como `public`.

Outra diferença entre classes em JavaScript e TypeScript é que, em TypeScript, não é necessário criar explicitamente um construtor para definir atributos na classe. Em JavaScript, se você quiser definir e inicializar os atributos de uma classe, precisa definir um método construtor para isso. No entanto, em TypeScript, os atributos podem ser declarados diretamente na classe, sem a necessidade de um construtor, tornando a sintaxe mais concisa e legível.

11.6. Conclusão

Este capítulo apresentou uma visão geral do TypeScript, destacando a tipagem estática opcional, os conceitos de POO, as interfaces, as classes e os modificadores de acesso.

No entanto, para explorar o potencial do TypeScript e utilizar suas funcionalidades de forma eficaz, é altamente recomendável que o leitor se aprofunde na [documentação oficial](https://www.typescriptlang.org/pt/)¹ ou procure outros recursos relacionados. Aprofundar-se no estudo de TypeScript não apenas solidificará o entendimento dos conceitos apresentados, mas também fornecerá percepções valiosas sobre práticas recomendadas e técnicas avançadas de desenvolvimento em TypeScript.

¹<https://www.typescriptlang.org/pt/>

11.7. Exercícios

1. Pesquise e explique as principais vantagens e desvantagens do TypeScript em relação ao JavaScript.
2. Faça uma pesquisa sobre os casos de uso do TypeScript.
3. Refaça os exercícios do capítulo 7 com TypeScript.
4. Refaça os exercícios do capítulo 8 com TypeScript.
5. Refaça os exercícios do capítulo 9 com TypeScript.
6. Refaça os exercícios do capítulo 10 com TypeScript.

12. Para Professores

Prezado(a) Professor(a),

Este livro foi cuidadosamente estruturado para facilitar o aprendizado de programação. O conteúdo foi decomposto e sequenciado no decorrer da obra de maneira gradual e lógica, com o intuito de oferecer a sequência e a dosagem adequadas de informações. Além disso, foram adicionados exercícios práticos em cada capítulo para integrar a teoria e a prática de forma contextualizada.

Todo o material complementar, incluindo exemplos de códigos e slides, está disponível neste repositório no GitHub: <https://github.com/jesielviana/livro-aprenda-programar-com-javascript>

12.1. E-book grátis para alunos

Os professores que adotarem este livro em suas disciplinas podem solicitar uma cópia em PDF para distribuir aos seus alunos. Para isso, basta entrar em contato com o autor: @jesielviana

13. Conclusão

Nesta conclusão, gostaria de parabenizar você, estimado leitor, pela jornada de aprendizado que você empreendeu ao longo deste livro. Ao absorver os conceitos fundamentais da programação, você construiu uma base sólida que a capacita a avançar para novos horizontes no mundo do desenvolvimento de software.

Agora, com as habilidades adquiridas, você está pronto para se aprofundar ainda mais e se tornar um profissional de desenvolvimento completo. Sugiro que você continue seus estudos, explorando tópicos como Programação Orientada a Objetos (POO), boas práticas de programação, desenvolvimento web e o conhecimento de frameworks populares.

Por fim, encorajo você a se aventurar por outras linguagens de programação, ampliando assim seu conjunto de habilidades e sua versatilidade como desenvolvedor. Lembre-se de que a jornada de aprendizado é contínua e que sempre haverá novos desafios e oportunidades de crescimento.

Este livro foi apenas o primeiro passo em sua jornada de desenvolvimento profissional, e estou confiante de que você está pronto para enfrentar os desafios que o futuro reserva. Continue explorando, aprendendo e construindo, e que seus caminhos na programação sejam repletos de sucesso e realização.

Obrigado por embarcar nesta jornada de aprendizado comigo, e desejo a você todo o melhor em suas futuras empreitadas no mundo da programação.

Com os melhores cumprimentos,

@jesielviana