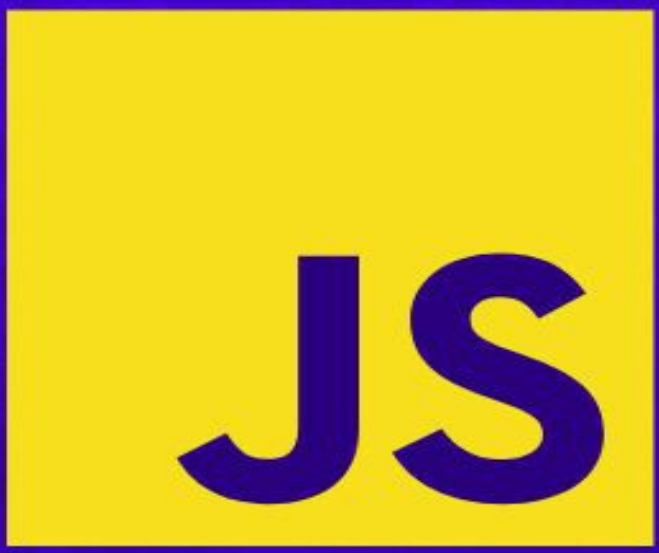
Com gabarito e explicações

200 exercícios de JavaScript



Explore todos os recursos de JS através de exercícios com solução e explicação

Matheus Battisti

Introdução

Seja bem-vindo ao ebook de 200 exercícios de JavaScript! Este livro é baseado em exercícios, ou seja, não teremos tanta teoria sendo apresentada. É um material de estudos para quem já possui conhecimentos básicos da linguagem e não completos iniciantes.

Se você deseja adquirir conhecimentos dos fundamentos de JavaScript, recomendo meu outro livro o: JavaScript para Iniciantes , ele é um bom ponto de partida para este aqui. Além disso, também ofereço um curso completo de JavaScript para você aprender do básico ao avançado.

No decorrer deste livro você encontrará a seguinte estrutura em todos os capítulos:

Enunciado do exercício, que é uma apresentação do problema a ser resolvido;
Código de solução;
Comentários a respeito da solução apresentada;

Os exercícios são distribuídos em diversos tópicos, como:

Sintaxe e Variáveis;
Tipos de Dados;
Operadores;
Controle de Fluxo (if, else, switch);
Laços de Repetição (for, while, do-while);
Funções;
Funções Arrow;
Funções de Alta Ordem (HOF) e Funções de Callback;
Template Strings;
Spread e Rest;
Destructuring;
Promises e Async/Await;
Recursos ES6+;
DOM e eventos;
e outros;

Os problemas foram distribuídos para agregar no aprendizado do leitor e também abordar os assuntos que são mais utilizados na programação de softwares em JavaScript.

O leitor também pode esperar por desafios e uma evolução gradual dos exercícios, tentando ao máximo refletir o dia a dia do programador.

E é claro, se está em busca de outros treinamentos, seja em JavaScript ou outra linguagem, confira também a nossa página de cursos da Hora de Codar . Lá você encontrará diversas opções, incluindo uma formação completa de Front-end , onde te ensino muito JavaScript!

Todos os nossos cursos possuem valores acessíveis, suporte para dúvidas, certificado de

conclusão, exercícios e projetos. Recursos que vão amplificar seu aprendizado.

Dada a introdução, vamos prosseguir?

Capítulo 1: Configuração do Ambiente de Desenvolvimento

Se você já tem o ambiente de desenvolvimento instalado, ou seja, o VS Code rodando na sua máquina e já lidou com JavaScript antes, pule este capítulo. Ele se destina a pessoas que precisam preparar o computador para poder resolver os exercícios.

Antes de começarmos a mergulhar no código, precisamos preparar nosso ambiente de trabalho. Ter o ambiente de desenvolvimento configurado corretamente é essencial para um processo de codificação eficiente e livre de estresse. Neste capítulo, vamos orientá-lo passo a passo para configurar um ambiente de desenvolvimento ideal para o JavaScript.

Falaremos sobre os melhores editores de texto para a codificação JavaScript, explicando seus pontos fortes e mostrando como tirar o máximo proveito deles. Além disso, vamos discutir a instalação e o uso de Node.js, uma ferramenta que permite executar JavaScript fora do navegador.

Também vamos introduzir o uso de ferramentas adicionais que podem aumentar sua produtividade, como os sistemas de controle de versão e extensões úteis.

Pronto para começar? Vamos configurar seu ambiente de desenvolvimento JavaScript!

Instalando e Configurando um Editor de Texto (como o VS Code)

Um editor de texto é uma ferramenta essencial para qualquer desenvolvedor. É onde você irá escrever, editar, e revisar seu código. Existem muitos editores de texto disponíveis, mas para este guia, vamos focar no Visual Studio Code (VS Code), uma ferramenta popular e robusta com recursos úteis para o desenvolvimento JavaScript.

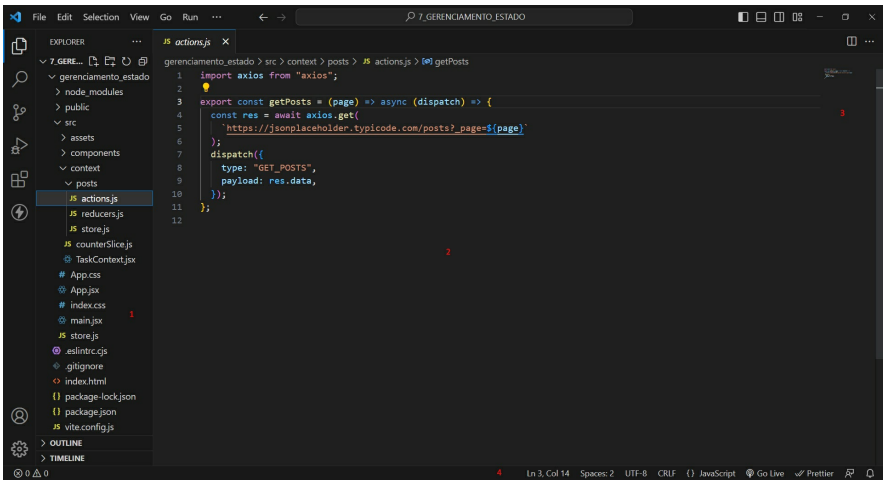
Passo 1: Instalação do Visual Studio Code

Acesse o site oficial do VS Code no endereço https://code.visualstudio.com .
Clique no botão "Download" para a versão adequada ao seu sistema operacional (Windows, macOS ou Linux).
Depois de baixar o arquivo, execute-o e siga as instruções na tela para concluir a instalação.

Passo 2: Familiarizando-se com o VS Code

Após a instalação, abra o VS Code. No início, você verá uma interface de usuário limpa e organizada, dividida em várias seções:

Barra de atividades(1): Localizada no lado esquerdo da janela. Aqui, você pode alternar entre diferentes vistas, como o explorador de arquivos, pesquisa, controle de versão e extensões.
Editor de código(2): A área central é a mais ampla, onde você vai escrever e editar seu código.
Painel de controle(3): Situado no lado direito, mostra detalhes e opções sobre o arquivo ou a seleção atual.
Barra de status(4): Na parte inferior, fornece informações sobre o projeto atual e permite que você acesse várias configurações e comandos.

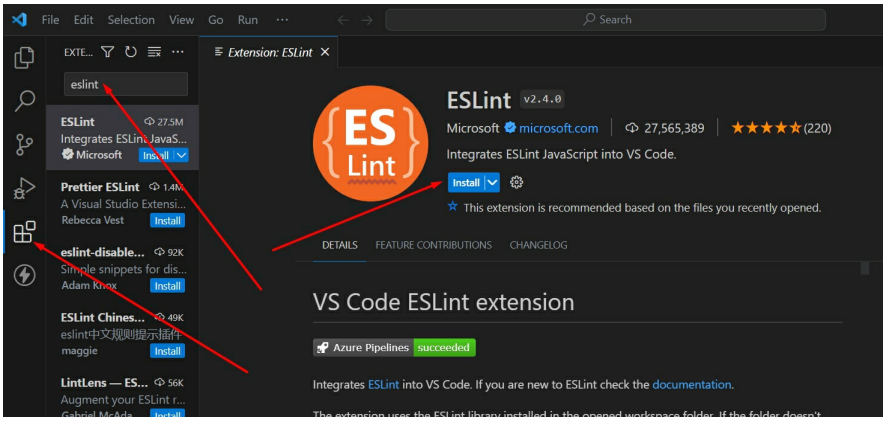


Na imagem acima as áreas descritas estão determinadas, note que o editor está diferente do seu pois há um projeto em andamento aberto.

Passo 3: Configurando o VS Code para JavaScript

O VS Code já vem com suporte para JavaScript por padrão, mas algumas extensões podem melhorar ainda mais sua experiência de codificação. Para instalar uma extensão, clique no ícone de extensões na barra de atividades e procure por estas extensões:

ESLint: Ajuda a identificar e corrigir problemas em seu código JavaScript.
Prettier: Um formatador de código que mantém seu código limpo e padronizado.
Visual Studio IntelliCode: Melhora a conclusão de código com inteligência artificial.
Para instalar, clique no botão "Install" na página da extensão. Uma vez instalada, a extensão estará pronta para uso.



Processo de instalação das extensões.

Agora, você tem o VS Code configurado para o desenvolvimento JavaScript. Na próxima seção, vamos falar sobre como instalar e configurar o Node.js.

Configurando o Console do Navegador

O console do navegador é uma ferramenta poderosa que permite testar e depurar seu código JavaScript. Aqui está um guia rápido sobre como acessar e usar o console do navegador.

Passo 1: Acessando o Console do Navegador

Praticamente todos os navegadores modernos têm ferramentas de desenvolvimento embutidas que incluem um console. Abaixo, estão as instruções para abrir o console nos navegadores mais populares:

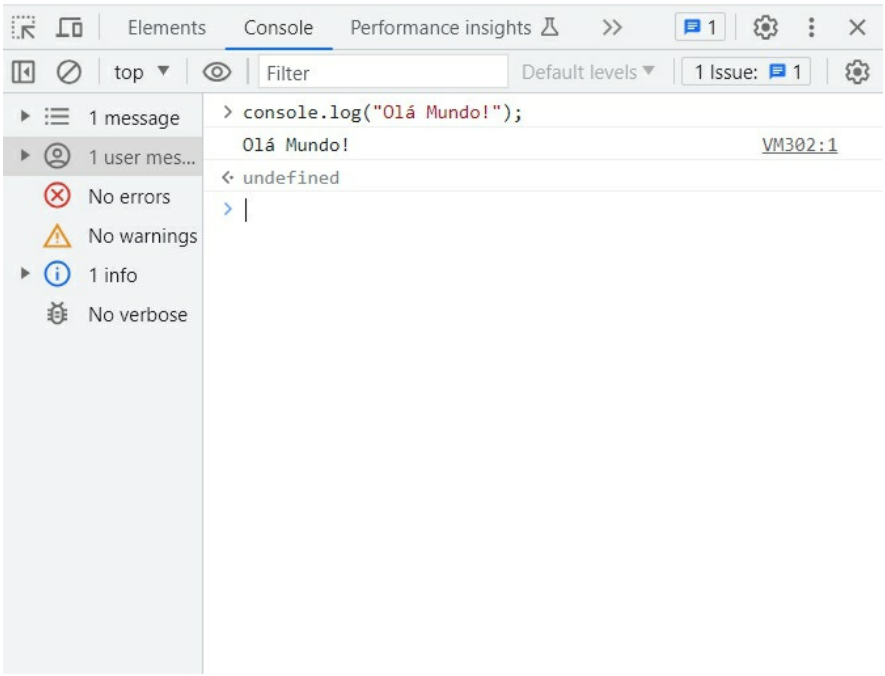
Google Chrome: Use o atalho Ctrl+Shift+J (Windows/Linux) ou Cmd+Option+J (Mac) ou clique com o botão direito do mouse em qualquer lugar da página, selecione "Inspecionar" e então escolha a aba "Console".
Firefox: Use o atalho Ctrl+Shift+K (Windows/Linux) ou Cmd+Option+K (Mac) ou clique com o botão direito do mouse em qualquer lugar na página, selecione "Inspecionar Elemento" e então escolha a aba "Console".
Safari: Primeiro, habilite o menu "Desenvolvedor" nas preferências do Safari. Depois, use o atalho Cmd+Option+C ou vá ao menu "Desenvolvedor" e escolha "Mostrar Console JavaScript".

Passo 2: Usando o Console do Navegador

O console do navegador é principalmente usado para:

Logar informações: Utilize o comando console.log() para imprimir valores e depurar seu código. Por exemplo, console.log("Olá, mundo!"); imprimirá "Olá, mundo!" no console.
Testar trechos de código: Você pode escrever e executar código JavaScript diretamente no console.
Monitorar erros: O console exibirá mensagens de erro quando algo der errado com seu código JavaScript. Essas mensagens geralmente incluem a natureza e a localização do erro.

Praticar e se familiarizar com o console do navegador será extremamente útil para o seu desenvolvimento com JavaScript.



Execução de código JavaScript no console do navegador.

Executando JavaScript no Navegador

Para rodar JavaScript diretamente em seu navegador, você precisará criar um arquivo HTML que referencia um arquivo JavaScript. Aqui está um passo a passo de como fazer isso.

Passo 1: Crie um Arquivo HTML

Primeiro, você precisará criar um arquivo HTML básico. Abra o seu editor de texto (VS Code, por exemplo), crie um novo arquivo e salve-o com a extensão .html (por exemplo, index.html). Em seguida, escreva um código HTML básico. Aqui está um exemplo:

```
<!DOCTYPE html>
< html >
< head >
   < title > Teste JavaScript </ title >
</ head >
< body >
   < h1 > Olá, Mundo! </ h1 >
</ body >
</ html >
```

Passo 2: Crie um Arquivo JavaScript

Agora, você precisará criar um arquivo JavaScript. No mesmo diretório do seu arquivo HTML, crie um novo arquivo e salve-o com a extensão .js (por exemplo, main.js). Neste arquivo, escreva algum código JavaScript. Por exemplo:

```
console .log( 'Olá, Console!' );
```

Passo 3: Conecte o Arquivo JavaScript ao HTML

Em seguida, você precisará referenciar o arquivo JavaScript em seu arquivo HTML. Para isso, adicione uma tag <script> com o atributo src apontando para o arquivo JavaScript, logo antes do fechamento da tag </body>. Aqui está como o seu HTML ficará:

```
<!DOCTYPE html>
< html >
< head >
   < title > Teste JavaScript </ title >
   < script  src= "main.js" ></ script >
</ head >
< body >
   < h1 > Olá, Mundo! </ h1 >
</ body >
</ html >
```

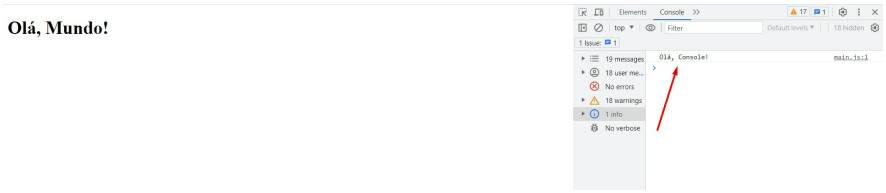
Passo 4: Abra o Arquivo HTML em seu Navegador

Finalmente, abra o arquivo HTML em seu navegador. Você pode fazer isso simplesmente navegando até o local do arquivo no seu sistema de arquivos e dando dois cliques no arquivo HTML.

O seu ambiente de desenvolvimento deve estar desta maneira.

Passo 5: Veja o Resultado no Console

Para ver o resultado do seu código JavaScript, abra o console do navegador. Você deve ver a mensagem 'Olá, Console!'. E pronto! Agora você sabe como executar código JavaScript em seu navegador.



Resultado final no navegador.

Este processo deve ser repetido em todos os exemplos de código e exercícios ao longo deste e-book. Organize os arquivos em uma pasta no seu computador para ser de fácil acesso.

Com o tempo, você aprenderá a usar o JavaScript para manipular elementos HTML, responder a eventos do usuário e muito mais!

Conclusão: Configuração do Ambiente de Desenvolvimento

Agora que configuramos o ambiente de desenvolvimento, estamos prontos para embarcar na fascinante jornada de programação com JavaScript. Configuramos um editor de texto, neste caso, o VS Code, que será uma ferramenta indispensável para escrever e editar nosso código. Aprendemos também a usar o console do navegador, uma ferramenta poderosa que nos permite interagir com páginas da web em tempo real e ver os resultados do nosso código JavaScript.

Por fim, demos os primeiros passos práticos ao criar e conectar um arquivo JavaScript a um arquivo HTML e executá-lo no navegador. Esses passos marcam o início de nossa exploração prática do JavaScript.

Embora essas configurações iniciais possam parecer simples, elas estabelecem a base sobre a qual construiremos uma compreensão sólida do JavaScript e suas capacidades. Lembre-se de que cada grande jornada começa com um pequeno passo.

Antes de irmos à prática, vou deixar algumas indicações. Desafios de JavaScript e 20+ Projetos de JavaScript são dois cursos que desenvolvi com uma abordagem totalmente prática, como este livro. Neles você também encontrará muitos desafios que te ajudarão a desenvolver suas habilidades em JavaScript.

Agora vamos aos exercícios!

Capítulo 2: Fundamentos de JavaScript

JavaScript é a linguagem de programação que traz vida às suas páginas da web. É responsável por adicionar interatividade, trabalhar com dados e muito mais.

Por isso, compreender seus fundamentos é um passo crucial para se tornar um desenvolvedor web competente.

Neste capítulo, trabalharemos os fundamentos da linguagem JavaScript, desde a sintaxe básica até os elementos mais usados em seu dia a dia como desenvolvedor.

Os tópicos cobertos incluem:

> **Sintaxe e Variáveis:** Vamos começar com o básico, declarando variáveis, atribuindo valores e imprimindo os resultados na console. Esse é o primeiro passo para começar a escrever qualquer código.
> **Tipos de Dados:** JavaScript possui vários tipos de dados que podem ser usados de diferentes maneiras. Vamos explorar alguns dos mais comuns e como eles funcionam.
> **Operadores:** Em seguida, entraremos nos operadores, que permitem realizar operações matemáticas, comparar valores e muito mais.
> **Controle de Fluxo:** Aqui, você aprenderá a controlar o fluxo do seu código usando estruturas condicionais como 'if', 'else' e 'switch'.
> **Laços de Repetição:** Por fim, mas não menos importante, exploraremos os laços de repetição. Isso permitirá que seu código execute uma série de instruções várias vezes, economizando tempo e esforço.

A cada seção, teremos uma série de exercícios práticos, começando do básico e aumentando gradativamente a dificuldade, permitindo que você aplique o que aprendeu.

Cada exercício irá desafiá-lo a escrever código, resolver problemas e, em seguida, analisar as soluções propostas, para garantir um entendimento completo do material.

Todos os exercícios deste e-book também estão disponíveis no GitHub:

Repositório do E-Book

Você pode fazer o download e ter acesso a este material no seu computador, ou também visualizar pelo navegador sempre que for necessário.

Está pronto? Então vamos!

Variáveis, tipos de dados e operadores

Nesta seção, estaremos nos aprofundando nos principais conceitos de JavaScript: variáveis, tipos de dados e operadores. As variáveis são o coração de qualquer programa, pois são responsáveis por armazenar e manipular os dados.

Os tipos de dados nos ajudam a entender o tipo de informação que estamos manipulando, seja ela um número, uma string, um booleano ou outros. Os operadores nos permitem realizar operações entre as variáveis e os valores, seja ela uma operação aritmética, uma comparação ou uma operação lógica.

Os exercícios propostos nesta seção irão permitir que você pratique e consolide seu conhecimento sobre esses três conceitos fundamentais. O objetivo é que, ao final desta seção, você seja capaz de declarar e manipular variáveis, entender e utilizar os diferentes tipos de dados, e utilizar operadores para realizar operações entre as variáveis e os valores.

Exercício 1: Declare duas variáveis chamadas "idade" e "peso", atribua valores a elas e imprima os valores na console.

Código de solução:

```
let idade = 25;
let peso = 70;

console.log("Idade: ", idade);
console.log("Peso: ", peso);
```

Explicação:
Este exercício envolve a declaração e inicialização de variáveis, e também a impressão dos valores dessas variáveis no console. Aqui, let é usado para declarar variáveis idade e peso, e então são atribuídos valores a elas. Em seguida, usamos o console.log() para imprimir os valores das variáveis.

Exercício 2: Declare uma constante chamada "PI", atribua o valor de Pi a

ela e imprima o valor na console.

Código de solução:

```
const  PI = 3.14159 ;
console .log( "PI: " , PI);
```

Explicação:

Neste exercício, estamos usando const para declarar uma constante. Diferente de uma variável declarada com let, uma constante não pode ser reatribuída depois de ser inicializada. Aqui, estamos atribuindo o valor de Pi à constante PI e, em seguida, usando console.log() para imprimir seu valor.

Exercício 3: Declare uma variável chamada "númeroMáximo", atribua a ela o valor máximo que um número pode ter em JavaScript e imprima o valor na console.

Código de solução:

```
let  númeroMáximo = Number .MAX_VALUE;
console .log( "Número Máximo: " , númeroMáximo);
```

Explicação:

JavaScript tem um valor máximo de número que pode ser representado, que é acessível através da propriedade estática Number.MAX_VALUE. Aqui, estamos atribuindo esse valor a uma variável chamada númeroMáximo e, em seguida, imprimindo-a usando console.log().

Exercício 4: Declare duas variáveis chamadas "nome" e "sobrenome", atribua o seu nome e sobrenome a elas. Concatene-as em uma terceira variável chamada "nomeCompleto" e imprima.

Código de solução:

```
let  nome = "João" ;
let  sobrenome = "Silva" ;
let  nomeCompleto = nome + " "  + sobrenome;

console .log( "Nome Completo: " , nomeCompleto);
```

Explicação:

Este exercício é sobre a concatenação de strings. Declaramos duas variáveis, nome e sobrenome, e as atribuímos a valores de string. Em seguida, declaramos uma terceira variável, nomeCompleto, e atribuímos a ela a concatenação de nome e sobrenome com um espaço entre elas. O operador + é usado para concatenar strings. Finalmente, usamos console.log() para imprimir o nomeCompleto.

Exercício 5: Declare duas variáveis e atribua valores numéricos a elas. Use o operador de adição para realizar operações entre as duas variáveis. Imprima o resultado.

Código de solução:

```
let num1 = 10 ;
let num2 = 20 ;

let soma = num1 + num2;

console .log( "Soma: " , soma);
```

Explicação:

Declaramos duas variáveis, num1 e num2, e as atribuímos a valores numéricos. Em seguida, declaramos uma terceira variável, soma, e atribuímos a ela a soma de num1 e num2 usando o operador de adição (+). Por fim, usamos o console.log() para imprimir o resultado.

Exercício 6: Declare duas variáveis e atribua valores booleanos a elas. Use os operadores lógicos AND, OR e NOT para realizar operações lógicas entre as variáveis. Imprima os resultados.

Lembrando que os operadores lógicos são: &&, || e !

Código de solução:

```
let bool1 = true ;
let bool2 = false ;

console .log( "AND: " , bool1 && bool2);
console .log( "OR: " , bool1 || bool2);
console .log( "NOT: " , !bool1);
```

Explicação:

Neste exercício, declaramos duas variáveis booleanas, bool1 e bool2, e realizamos

operações lógicas entre elas. Usamos o operador lógico AND (&&) para imprimir o resultado de bool1 && bool2, o operador OR (||) para imprimir o resultado de bool1 || bool2, e o operador NOT (!) para imprimir o resultado de !bool1.

Exercício 7: Declare três variáveis e atribua valores numéricos a elas. Use operadores de comparação para comparar os valores entre as variáveis. Imprima os resultados.

Exemplos de operadores de comparação: >, <, >=, <= e outros.

Código de solução:

```
let num1 = 10;
let num2 = 20;
let num3 = 30;

console.log("num1 < num2: ", num1 < num2);
console.log("num2 > num3: ", num2 > num3);
console.log("num1 == num3: ", num1 == num3);
```

Explicação:

Neste exercício, declaramos três variáveis numéricas, num1, num2 e num3. Em seguida, usamos operadores de comparação para comparar os valores entre elas e imprimimos os resultados. Usamos o operador < para verificar se num1 é menor que num2, o operador > para verificar se num2 é maior que num3 e o operador == para verificar se num1 é igual a num3.

Exercício 8: Declare uma variável e atribua um valor numérico a ela. Use o operador de incremento para aumentar o valor da variável. Imprima o resultado.

Código de solução:

```
let num = 10;
num++;

console.log("Número incrementado: ", num);
```

Explicação:

Neste exercício, declaramos uma variável num e a atribuímos a um valor numérico. Usamos o operador de incremento ++ para aumentar o valor de num em 1. Por fim, usamos console.log() para imprimir o resultado.

Operadores Condicionais

Na seção de operadores condicionais, nosso objetivo é compreender e aplicar os conceitos de controle de fluxo em JavaScript. As estruturas de controle de fluxo são um dos principais pilares de qualquer linguagem de programação, pois elas nos permitem controlar o fluxo de execução do nosso código.

Nós vamos nos concentrar nos três principais operadores condicionais: if, else e switch. Vamos explorar como esses operadores podem ser usados para implementar lógicas condicionais no nosso código, permitindo que nosso programa tome decisões com base em certas condições.

Os exercícios desta seção vão desde a verificação básica de condições até cenários mais complexos que envolvem a combinação de vários operadores condicionais. Através destes exercícios, você desenvolverá a habilidade de implementar e gerenciar o fluxo de execução do seu código com eficácia.

Exercício 9: Escreva um programa que recebe um número e verifica se ele é positivo, negativo ou zero.

Descrição: Você precisará de uma função que receba um número como argumento. Essa função usará estruturas if, else if e else para determinar e imprimir se o número é positivo, negativo ou zero. Por exemplo, se recebermos o número 10, a saída deve ser "positivo". Se recebermos -5, a saída deve ser "negativo". Se recebermos 0, a saída deve ser "zero".

Código de solução:

```
function  verificarNumero (num) {
    if (num > 0 ) {
        console .log( "O número é positivo" );
   } else  if (num < 0 ) {
        console .log( "O número é negativo" );
   } else  {
        console .log( "O número é zero" );
   }
}

verificarNumero( 10 );   // Imprime: O número é positivo
verificarNumero( -5 );   // Imprime: O número é negativo
verificarNumero( 0 );   // Imprime: O número é zero
```

Explicação: A função verificarNumero verifica se o número passado como argumento é maior que 0, menor que 0 ou igual a 0, usando uma estrutura de controle if/else

if/else. Se o número for maior que 0, ela imprime "O número é positivo". Se for menor que 0, imprime "O número é negativo". Se for igual a 0, imprime "O número é zero". Ao chamar a função com os números 10, -5 e 0, podemos ver essa lógica em ação.

Exercício 10: Escreva um programa que recebe duas notas de um aluno, calcula a média e imprime se o aluno foi aprovado ou reprovado (considerando que a média para aprovação é 7).

Descrição: Nesse exercício, você vai criar uma função que recebe duas notas, calcula a média e, com base na média, decide se o aluno foi aprovado (média maior ou igual a 7) ou reprovado (média menor que 7). Por exemplo, se as notas forem 8 e 6, a média será 7 e o aluno será aprovado.

Código de solução:

```
function  calcularMedia (nota1, nota2) {
    var  media = (nota1 + nota2) / 2 ;
    if (media >= 7 ) {
        console .log( "Aluno aprovado com a média: "  + media);
   } else  {
        console .log( "Aluno reprovado com a média: "  + media);
   }
}

calcularMedia( 8 , 6 );   // Imprime: Aluno aprovado com a média: 7
calcularMedia( 5 , 6 );   // Imprime: Aluno reprovado com a média: 5.5
```

Explicação: A função calcularMedia recebe duas notas como argumentos, calcula a média dessas notas e, então, usa uma estrutura de controle if/else para determinar se o aluno foi aprovado ou reprovado. Se a média for maior ou igual a 7, a função imprime "Aluno aprovado com a média: " seguido do valor da média. Se a média for menor que 7, a função imprime "Aluno reprovado com a média: " seguido do valor da média. Quando chamamos a função com as notas 8 e 6, a média é 7 e, portanto, o aluno é aprovado. Quando chamamos a função com as notas 5 e 6, a média é 5.5 e, portanto, o aluno é reprovado.

Exercício 11: Escreva um programa que verifica se uma pessoa pode votar com base na idade.

Descrição: Nesse exercício, você vai criar uma função que recebe a idade de uma pessoa e verifica se ela pode votar. No Brasil, o voto é obrigatório para pessoas entre 18 e 70 anos, facultativo para pessoas com 16 ou 17 anos ou acima de 70. Menores de 16 anos não votam.

Código de solução:

```
function  podeVotar (idade) {
    if (idade >= 18  && idade < 70 ) {
        console .log( "Voto obrigatório." );
   } else  if ((idade >= 16  && idade < 18 ) || idade >= 70 ) {
        console .log( "Voto facultativo." );
   } else  {
        console .log( "Não vota." );
   }
}

podeVotar( 15 );   // Imprime: Não vota.
podeVotar( 16 );   // Imprime: Voto facultativo.
podeVotar( 18 );   // Imprime: Voto obrigatório.
podeVotar( 70 );   // Imprime: Voto facultativo.
```

Explicação: A função podeVotar recebe uma idade como argumento e, então, usa uma estrutura de controle if/else if/else para determinar a obrigatoriedade do voto para essa idade. Se a idade for maior ou igual a 18 e menor que 70, a função imprime "Voto obrigatório.". Se a idade for maior ou igual a 16 e menor que 18, ou maior ou igual a 70, a função imprime "Voto facultativo.". Caso contrário, a função imprime "Não vota.". Quando chamamos a função com as idades 15, 16, 18 e 70, podemos ver essa lógica em ação.

Exercício 12: Escreva um programa que verifica a situação de um estudante de acordo com sua média final.

Descrição: Neste exercício, você vai criar uma função que recebe a média final de um estudante e verifica sua situação, se aprovado (média igual ou superior a 7), se em recuperação (média entre 5 e 6.9) ou se reprovado (média abaixo de 5).

Código de solução:

```
function  situacaoEstudante (mediaFinal) {
    if (mediaFinal >= 7 ) {
        console .log( "Aprovado." );
   } else  if (mediaFinal >= 5  && mediaFinal < 7 ) {
        console .log( "Em recuperação." );
   } else  {
        console .log( "Reprovado." );
   }
}

situacaoEstudante( 7 );   // Imprime: Aprovado.
situacaoEstudante( 6 );   // Imprime: Em recuperação.
```

```
situacaoEstudante( 4 );   // Imprime: Reprovado.
```

Explicação: A função situacaoEstudante recebe uma média final como argumento e, então, usa uma estrutura de controle if/else if/else para determinar a situação do estudante com base nessa média. Se a média for maior ou igual a 7, a função imprime "Aprovado.". Se a média for maior ou igual a 5 e menor que 7, a função imprime "Em recuperação.". Caso contrário, a função imprime "Reprovado.". Quando chamamos a função com as médias 7, 6 e 4, podemos ver essa lógica em ação.

Exercício 13: Escreva um programa que calcula o IMC (Índice de Massa Corporal) de uma pessoa e imprime uma mensagem indicando se a pessoa está abaixo do peso, com peso normal, com sobrepeso ou obesa.

Descrição: Nesse exercício, você vai criar uma função que recebe o peso (em kg) e a altura (em m) de uma pessoa, calcula o IMC e verifica em qual faixa o valor se encaixa. As faixas são: abaixo do peso (IMC < 18.5), normal (18.5 ≤ IMC < 25), sobrepeso (25 ≤ IMC < 30) e obesidade (IMC ≥ 30).

Código de solução:

```
function  calcularIMC (peso, altura) {
    var  imc = peso / (altura * altura);
    if (imc < 18.5 ) {
        console .log( "Abaixo do peso" );
   } else  if (imc >= 18.5  && imc < 25 ) {
        console .log( "Peso normal" );
   } else  if (imc >= 25  && imc < 30 ) {
        console .log( "Sobrepeso" );
   } else  {
        console .log( "Obesidade" );
   }
}

calcularIMC( 60 , 1.7 );   // Imprime: Peso normal
calcularIMC( 80 , 1.7 );   // Imprime: Sobrepeso
calcularIMC( 90 , 1.7 );   // Imprime: Obesidade
```

Explicação: A função calcularIMC recebe peso e altura como argumentos, calcula o IMC e, então, usa uma estrutura de controle if/else if/else para determinar a faixa de peso na qual a pessoa se encaixa. Se o IMC for menor que 18.5, a função imprime "Abaixo do peso". Se o IMC for maior ou igual a 18.5 e menor que 25, a função imprime "Peso normal". Se o IMC for maior ou igual a 25 e menor que 30, a função imprime "Sobrepeso". Se o IMC for maior ou igual a 30, a função imprime "Obesidade". Quando chamamos a função com os pares de peso e altura (60, 1.7), (80, 1.7) e (90, 1.7), vemos essa lógica em ação.

Exercício 14: Escreva um programa que verifica se um ano é bissexto.

Descrição: Nesse exercício, você vai criar uma função que recebe um ano e verifica se ele é bissexto. Um ano é bissexto se for divisível por 4, exceto os que são divisíveis por 100 mas não por 400.

Código de solução:

```
function  verificarBissexto(ano) {
    if((ano % 4  == 0  && ano % 100  != 0 ) || (ano % 400  == 0 )) {
        console.log(ano + " é um ano bissexto" );
   } else  {
        console.log(ano + " não é um ano bissexto" );
   }
}

verificarBissexto( 2000 );   // Imprime: 2000 é um ano bissexto
verificarBissexto( 2001 );   // Imprime: 2001 não é um ano bissexto
verificarBissexto( 2100 );   // Imprime: 2100 não é um ano bissexto
```

Explicação: A função verificarBissexto recebe um ano como argumento e, então, usa uma estrutura de controle if/else para verificar se o ano é bissexto. Se o ano for divisível por 4 e não for divisível por 100, ou se o ano for divisível por 400, então é um ano bissexto e a função imprime o ano seguido da frase "é um ano bissexto". Caso contrário, a função imprime o ano seguido da frase "não é um ano bissexto". Quando chamamos a função com os anos 2000, 2001 e 2100, vemos essa lógica em ação.

Exercício 15: Escreva um programa que verifica se uma palavra é um palíndromo.

Descrição: Um palíndromo é uma palavra que tem a propriedade de poder ser lida tanto da direita para a esquerda como da esquerda para a direita. Nesse exercício, você irá criar uma função que recebe uma palavra como argumento e verifica se ela é um palíndromo.

Código de solução:

```
function  verificarPalindromo(palavra) {
    var  palavraInvertida = palavra.split( '' ).reverse().join( '' );
    if(palavra == palavraInvertida) {
        console.log(palavra + " é um palíndromo" );
   } else  {
        console.log(palavra + " não é um palíndromo" );
   }
}
```

```
verificarPalindromo( "arara" );   // Imprime: arara é um palíndromo
verificarPalindromo( "gato" );   // Imprime: gato não é um palíndromo
```

Explicação: A função verificarPalindromo recebe uma palavra como argumento e, então, cria uma nova string que é a palavra invertida, usando os métodos de string split, reverse e join. Se a palavra original for igual à palavra invertida, então a função imprime a palavra seguida da frase "é um palíndromo". Caso contrário, a função imprime a palavra seguida da frase "não é um palíndromo". Quando chamamos a função com as palavras "arara" e "gato", vemos essa lógica em ação.

Exercício 16: Escreva um programa que determina o maior entre três números.

Descrição: Nesse exercício, você irá criar uma função que recebe três números como argumentos e imprime o maior deles.

Código de solução:

```
function  maiorNumero (n1, n2, n3) {
    if (n1 > n2 && n1 > n3) {
        console .log(n1 + " é o maior número" );
   } else  if (n2 > n1 && n2 > n3) {
        console .log(n2 + " é o maior número" );
   } else  {
        console .log(n3 + " é o maior número" );
   }
}

maiorNumero( 1 , 2 , 3 );   // Imprime: 3 é o maior número
maiorNumero( 5 , 3 , 4 );   // Imprime: 5 é o maior número
maiorNumero( 7 , 8 , 7 );   // Imprime: 8 é o maior número
```

Explicação: A função maiorNumero recebe três números como argumentos. Se o primeiro número for maior que o segundo e o terceiro, a função imprime o primeiro número seguido da frase "é o maior número". Se o segundo número for maior que o primeiro e o terceiro, a função imprime o segundo número seguido da frase "é o maior número". Caso contrário, a função imprime o terceiro número seguido da frase "é o maior número". Quando chamamos a função com os conjuntos de números (1, 2, 3), (5, 3, 4) e (7, 8, 7), vemos essa lógica em ação.

Exercício 17: Escreva um programa que determina se um número é par ou ímpar.

Descrição: Nesse exercício, você vai criar uma função que recebe um número como

argumento e imprime se ele é par ou ímpar.

Código de solução:

```
function  parOuImpar (numero) {
    if (numero % 2  == 0 ) {
        console .log(numero + " é par" );
   } else  {
        console .log(numero + " é ímpar" );
   }
}

parOuImpar( 2 );   // Imprime: 2 é par
parOuImpar( 3 );   // Imprime: 3 é ímpar
```

Explicação: A função parOuImpar recebe um número como argumento. Se o resto da divisão desse número por 2 for igual a zero (o que significa que ele é divisível por 2), então a função imprime o número seguido da frase "é par". Caso contrário, a função imprime o número seguido da frase "é ímpar". Quando chamamos a função com os números 2 e 3, vemos essa lógica em ação.

Exercício 18: Escreva um programa que verifica se uma pessoa pode dirigir com base na idade.

Descrição: Nesse exercício, você vai criar uma função que recebe a idade de uma pessoa e verifica se ela pode dirigir. No Brasil, apenas pessoas com mais de 18 anos podem tirar carteira de habilitação.

Código de solução:

```
function  podeDirigir (idade) {
    if (idade >= 18) {
        console .log( "Pode dirigir." );
   } else  {
        console .log( "Não pode dirigir" );
   }
}

podeDirigir( 15 );   // Imprime: Não pode dirigir
podeDirigir( 18 );   // Imprime: Pode dirigir
podeDirigir( 36 );   // Imprime: Pode dirigir
```

Explicação: A função podeDirigir recebe uma idade como argumento e, então, usa uma estrutura de controle if/else para determinar se a pessoa está apta a dirigir. Se a idade for maior ou igual a 18, a função imprime "Pode dirigir.". Caso contrário, a função imprime "Não pode dirigir.". Quando chamamos a função com as idades 15, 18 e 36, podemos ver essa lógica em ação.

Exercício 19: Escreva um programa que verifica se um número está dentro de um determinado intervalo.

Descrição: Neste exercício, você vai criar uma função que recebe três números como argumentos: um número a ser verificado, o limite inferior e o limite superior do intervalo. A função deve imprimir se o número está dentro do intervalo ou não.

Código de solução:

```
function  dentroDoIntervalo(numero, limiteInferior, limiteSuperior) {
    if(numero >= limiteInferior && numero <= limiteSuperior) {
        console.log(numero + " está dentro do intervalo");
   } else {
        console.log(numero + " está fora do intervalo");
   }
}

dentroDoIntervalo( 5, 1, 10);   // Imprime: 5 está dentro do intervalo
dentroDoIntervalo( 15, 1, 10);   // Imprime: 15 está fora do intervalo
```

Explicação: A função dentroDoIntervalo recebe três números como argumentos: um número a ser verificado e dois limites que definem o intervalo. Se o número estiver dentro do intervalo (ou seja, maior ou igual ao limite inferior e menor ou igual ao limite superior), a função imprime o número seguido da frase "está dentro do intervalo". Caso contrário, imprime o número seguido da frase "está fora do intervalo". Quando chamamos a função com os argumentos (5, 1, 10) e (15, 1, 10), vemos essa lógica em ação.

Exercício 20: Escreva um programa que imprime uma classificação baseada na nota de um aluno.

Descrição: Neste exercício, você vai criar uma função que recebe uma nota de um aluno como argumento e imprime uma classificação com base nessa nota.

Código de solução:

```
function  classificarAluno(nota) {
    if(nota >= 90) {
        console.log("A");
   } else  if(nota >= 80) {
        console.log("B");
   } else  if(nota >= 70) {
        console.log("C");
   } else  if(nota >= 60) {
        console.log("D");
```

```
    } else {
        console.log( "F" );
    }
}

classificarAluno( 95 );   // Imprime: A
classificarAluno( 82 );   // Imprime: B
classificarAluno( 74 );   // Imprime: C
classificarAluno( 62 );   // Imprime: D
classificarAluno( 55 );   // Imprime: F
```

Explicação: A função classificarAluno recebe uma nota de aluno como argumento e imprime uma classificação baseada nessa nota. Se a nota for 90 ou superior, a função imprime "A". Se a nota for 80 ou superior (mas menor que 90), a função imprime "B". Se a nota for 70 ou superior (mas menor que 80), a função imprime "C". Se a nota for 60 ou superior (mas menor que 70), a função imprime "D". Se a nota for menor que 60, a função imprime "F". Quando chamamos a função com as notas 95, 82, 74, 62 e 55, vemos essa lógica em ação.

Exercício 21: Escreva um programa que determina a estação do ano com base no mês.

Descrição: Neste exercício, você criará uma função que recebe um número de 1 a 12 (representando os meses de janeiro a dezembro) e imprimirá a estação do ano correspondente.

Código de solução:

```
function determinarEstacao(mes) {
    if (mes < 3 || mes === 12 ) {
        console.log( "Verão" );
    } else if (mes < 6 ) {
        console.log( "Outono" );
    } else if (mes < 9 ) {
        console.log( "Inverno" );
    } else {
        console.log( "Primavera" );
    }
}

determinarEstacao( 12 );   // Imprime: Verão
determinarEstacao( 4 );   // Imprime: Outono
determinarEstacao( 7 );   // Imprime: Inverno
determinarEstacao( 10 );   // Imprime: Primavera
```

Explicação: A função determinarEstacao recebe um número representando um mês como argumento. Se o número for menor que 3 ou igual a 12 (janeiro, fevereiro ou dezembro), a função imprime "Verão". Se o número for menor que 6 (março, abril ou

maio), a função imprime "Outono". Se o número for menor que 9 (junho, julho ou agosto), a função imprime "Inverno". Caso contrário (setembro, outubro ou novembro), a função imprime "Primavera".

Exercício 22: Escreva um programa que determina o número de dias em um mês.

Descrição: Neste exercício, você criará uma função que recebe o número de um mês e imprime o número de dias desse mês.

Código de solução:

```
function  determinarDiasNoMes(mes) {
    switch(mes) {
        case  2:
            console.log("28 ou 29 dias");
            break;
        case  4:
        case  6:
        case  9:
        case  11:
            console.log("30 dias");
            break;
        default:
            console.log("31 dias");
   }
}

determinarDiasNoMes(2);   // Imprime: 28 ou 29 dias
determinarDiasNoMes(4);   // Imprime: 30 dias
determinarDiasNoMes(1);   // Imprime: 31 dias
```

Explicação: A função determinarDiasNoMes usa uma estrutura de controle switch para verificar o número de dias em um mês. O mês de fevereiro (2) pode ter 28 ou 29 dias, os meses de abril (4), junho (6), setembro (9) e novembro (11) têm 30 dias e todos os outros meses têm 31 dias.

Exercício 23: Escreva um programa que verifica as cores de um semafáro.

Descrição: Neste exercício, você deve criar uma função que recebe uma cor, e imprime no console o que essa cor representa, o console deve imprimir uma mensagem específica para cores que não existem em um semáforo..

Código de solução:

```
function  verificaSemafaro(cor)  {
    if  (cor === "verde") {
        console .log( "Siga em frente." );
   } else  if  (cor === "amarelo") {
        console .log( "Atenção, diminua a velocidade." );
   } else  if  (cor === "vermelho") {
        console .log( "Pare seu veículo." );
   } else  {
        console .log( "Envie uma cor válida." );
   }
}

verificaSemafaro( "verde" );   // Imprime: Siga em frente
verificaSemafaro( "amarelo" );   // Imprime: Atenção, diminua a velocidade
verificaSemafaro( "vermelho" );   // Imprime: Pare seu veículo
verificaSemafaro( "azul" );   // Impime: Envie uma cor válida
```

Explicação: A função varificaSemafaro recebe uma cor como argumento. Se a cor for igual a verde, a função imprime 'Siga em frente.'. Se a cor for igual a amarelo, a função imprime 'Atenção, diminua a velocidade.'. Se a cor for igual a vermelho, a função imprime 'Pare seu veículo.'. Por fim, se for uma outra cor, a função imprime 'Envie uma cor válida.'

Laços de Repetição

Os laços de repetição, também conhecidos como loops, são uma das principais ferramentas em programação, permitindo que um bloco de código seja repetido diversas vezes até que uma condição seja satisfeita. Eles são essenciais para a resolução de uma grande variedade de problemas de programação, como a repetição de uma ação, a iteração em uma coleção de itens ou a espera por uma condição ser satisfeita.

Em JavaScript, temos três estruturas principais de laços de repetição: for, while e do-while. Cada uma tem suas particularidades e usos ideais, mas todas elas permitem repetir um bloco de código baseado em uma condição.

Nesta seção, vamos explorar estes três tipos de laços através de vários exercícios práticos e variados. Os exercícios foram planejados para abranger uma ampla gama de problemas, desde a impressão de uma sequência de números até a criação de pequenos jogos e operações com matrizes. Além disso, vamos também aprender a manipular e transformar dados, como converter números decimais em binários e vice-versa.

Exercício 24: Usando um laço for, imprima os números de 1 a 10.

Descrição: Neste exercício, você deve criar um loop usando a estrutura de repetição for que percorra de 1 a 10 e imprima cada número no console.

Código de solução:

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

Explicação: Este código inicia uma variável i em 1 e, enquanto i for menor ou igual a 10, imprime i no console e depois incrementa i em 1. O laço for é especialmente útil quando sabemos exatamente quantas vezes queremos que nosso código seja executado.

Exercício 25: Usando um laço while, imprima os números de 10 a 1 (em ordem decrescente).

Descrição: Neste exercício, você deve criar um loop usando a estrutura de repetição while que percorra de 10 a 1 (em ordem decrescente) e imprima cada número no console.

Código de solução:

```
let  i = 10 ;
while (i >= 1 ){
    console .log(i);
   i--;
}
```

Explicação: Este código inicia uma variável i em 10 e, enquanto i for maior ou igual a 1, imprime i no console e depois decrementa i em 1. O laço while é útil quando não sabemos exatamente quantas vezes o código precisa ser executado, mas temos uma condição de parada definida.

Exercício 26: Usando um laço do-while, imprima todos os números ímpares de 1 a 20.

Descrição: Neste exercício, você deve criar um loop usando a estrutura de repetição do-while que percorra todos os números de 1 a 20 e imprima no console apenas os números ímpares.

Código de solução:

```
let  i = 1 ;
do  {
```

```
    if (i % 2  != 0 ){
        console .log(i);
   }
   i++;
} while (i <= 20 );
```

Explicação: Este código inicia uma variável i em 1 e, enquanto i for menor ou igual a 20, verifica se i é ímpar (o resto da divisão de i por 2 é diferente de zero) e, se for, imprime i no console. Depois, incrementa i em 1. O laço do-while garante que o bloco de código seja executado pelo menos uma vez, antes de verificar a condição.

Exercício 27: Usando um laço for, imprima a tabuada de multiplicação do número 5.

Descrição: Neste exercício, você deve criar um loop usando a estrutura de repetição for que percorra de 1 a 10 e imprima a tabuada de multiplicação do número 5.

Código de solução:

```
for ( let  i = 1 ; i <= 10 ; i++){
    console .log( "5 x "  + i + " = "  + 5 *i);
}
```

Explicação: Este código inicia uma variável i em 1 e, enquanto i for menor ou igual a 10, imprime a multiplicação de 5 por i no console e depois incrementa i em 1. Usamos o laço for, pois sabemos exatamente quantas vezes nosso código precisa ser executado (10 vezes).

Exercício 28: Usando um laço while, calcule e imprima a soma de todos os números de 1 a 100.

Descrição: Neste exercício, você deve criar um loop usando a estrutura de repetição while que percorra de 1 a 100 e calcule a soma de todos esses números.

Código de solução:

```
let  i = 1 ;
let  soma = 0 ;
while (i <= 100 ){
   soma += i;
   i++;
}
console .log(soma);
```

Explicação: Este código inicia uma variável i em 1 e uma variável soma em 0. Enquanto i for menor ou igual a 100, adiciona i à soma e incrementa i em 1. Ao final, imprime a soma no console. O laço while é útil quando não sabemos exatamente quantas vezes o código precisa ser executado, mas temos uma condição de parada definida.

Exercício 29: Usando um laço do-while, imprima todos os números pares de 1 a 100.

Descrição: Neste exercício, você deve criar um loop usando a estrutura de repetição do-while que percorra todos os números de 1 a 100 e imprima no console apenas os números pares.

Código de solução:

```
let  i = 1 ;
do  {
    if (i % 2  == 0 ){
        console .log(i);
   }
   i++;
} while (i <= 100 );
```

Explicação: Este código inicia uma variável i em 1 e, enquanto i for menor ou igual a 100, verifica se i é par (o resto da divisão de i por 2 é igual a zero) e, se for, imprime i no console. Depois, incrementa i em 1. O laço do-while garante que o bloco de código seja executado pelo menos uma vez, antes de verificar a condição.

Exercício 30: Usando um laço for, imprima todos os números primos de 1 a 100.

Descrição: Neste exercício, você precisa percorrer todos os números de 1 a 100 e imprimir apenas aqueles que são primos. Lembre-se, um número primo é um número natural maior do que 1 que não tem divisores positivos outros que 1 e ele mesmo.

Código de solução:

```
for ( let  num = 2 ; num <= 100 ; num++) {
    let  isPrime = true ;
    for ( let  div = 2 ; div < num; div++) {
        if (num % div == 0 ){
           isPrime = false ;
            break ;
       }
```

```
    }
     if (isPrime){
         console .log(num);
    }
}
```

Explicação: Aqui temos um laço for externo que percorre os números de 2 a 100 e um laço for interno que verifica se algum número entre 2 e o número em questão (excluindo-o) é divisor desse número. Se encontrar algum divisor, a variável isPrime é definida como false e o laço interno é interrompido. Se nenhum divisor for encontrado, isPrime permanecerá true e o número será impresso.

Exercício 31: Usando um laço while, crie um programa que adivinha um número que o usuário pensou, através do método de busca binária.

Descrição: Neste exercício, você vai simular um jogo onde o usuário "pensa" em um número de 1 a 100 e o programa deve adivinhar o número usando o método de busca binária. Para simplificar, vamos supor que o número pensado pelo usuário é 50.

Código de solução:

```
let  min = 1 ;
let  max = 100 ;
let  chute = Math .floor((max + min)/ 2 );
let  numeroPensado = 50 ;

while (chute !== numeroPensado){
    if (chute > numeroPensado){
       max = chute;
   }
    else {
       min = chute;
   }
   chute = Math .floor((max + min)/ 2 );
}
console .log( "O número pensado é: "  + chute);
```

Explicação: Este código inicia com os valores mínimo e máximo possíveis para o número pensado (1 e 100) e faz um "chute" no meio do intervalo. Enquanto o chute for diferente do número pensado, ajusta o valor mínimo ou máximo de acordo com o chute e faz um novo chute no meio do intervalo atualizado. Quando o chute é igual ao número pensado, o laço termina e o número é impresso.

Exercício 32: Usando um laço do-while, crie um jogo de "Pedra, papel e tesoura" contra o computador.

Descrição: Neste exercício, você vai simular um jogo de "Pedra, papel e tesoura" contra o computador. Para simplificar, vamos supor que a escolha do usuário é sempre "pedra".

Código de solução:

```
let escolhaUsuario = "pedra";
let escolhaComputador;
let resultado;
do {
    let aleatorio = Math.floor(Math.random() * 3);
    switch (aleatorio){
        case 0:
           escolhaComputador = "pedra";
            break;
        case 1:
           escolhaComputador = "papel";
            break;
        default:
           escolhaComputador = "tesoura";
   }

    if ((escolhaUsuario === "pedra" && escolhaComputador === "tesoura") ||
   (escolhaUsuario === "papel" && escolhaComputador === "pedra") ||
   (escolhaUsuario === "tesoura" && escolhaComputador === "papel")){
       resultado = "Usuário ganhou!";
   }
    else if (escolhaUsuario === escolhaComputador){
       resultado = "Empate!";
   }
    else {
       resultado = "Computador ganhou!";
   }
    console.log("Usuário: " + escolhaUsuario + " - Computador: " +
escolhaComputador + " - Resultado: " + resultado);
} while (resultado === "Empate!");
```

Explicação: Este código inicia com a escolha do usuário ("pedra") e escolhe uma opção aleatória para o computador ("pedra", "papel" ou "tesoura"). Depois, verifica qual opção ganha de acordo com as regras do jogo e define o resultado ("Usuário

Exercício 33: Usando um laço for, imprima os primeiros 10 números da sequência de Fibonacci.

Descrição: Neste exercício, você precisa imprimir os primeiros 10 números da

sequência de Fibonacci. A sequência de Fibonacci é uma sequência de números onde cada número subsequente é a soma dos dois anteriores. Ela começa assim: 0, 1, 1, 2, 3, 5, 8, 13...

Código de solução:

```
let  num1 = 0 ;
let  num2 = 1 ;
console .log(num1);
console .log(num2);
for ( let  i = 2 ; i < 10 ; i++) {
    let  nextNum = num1 + num2;
    console .log(nextNum);
   num1 = num2;
   num2 = nextNum;
}
```

Explicação: Este código inicia com os dois primeiros números da sequência de Fibonacci (0 e 1) e os imprime. Depois, entra em um laço for que será executado 8 vezes (para imprimir os próximos 8 números da sequência). A cada iteração do laço, calcula o próximo número como a soma dos dois anteriores, imprime esse número e atualiza os dois números anteriores.

Exercício 34: Usando um laço while, imprima os números de 1 a 100, mas para múltiplos de 3 imprima "Fizz" e para múltiplos de 5 imprima "Buzz". Para números que são múltiplos de ambos, imprima "FizzBuzz" (Problema FizzBuzz).

Descrição: Neste exercício, você precisa imprimir os números de 1 a 100, mas com uma condição especial. Se um número for múltiplo de 3, deve imprimir "Fizz" em vez do número. Se for múltiplo de 5, deve imprimir "Buzz". E se for múltiplo de ambos (3 e 5), deve imprimir "FizzBuzz".

Código de solução:

```
let  num = 1 ;
while (num <= 100 ) {
    if (num % 3  === 0  && num % 5  === 0 ) {
        console .log( "FizzBuzz" );
   }
    else  if (num % 3  === 0 ) {
        console .log( "Fizz" );
   }
    else  if (num % 5  === 0 ) {
        console .log( "Buzz" );
   }
    else  {
```

```
        console.log(num);
    }
    num++;
}
```

Explicação: Este código inicia com um número igual a 1 e entra em um laço while que continuará até que o número seja maior que 100. Dentro do laço, verifica se o número atual é múltiplo de 3 e 5 (imprimindo "FizzBuzz" se for), múltiplo de 3 (imprimindo "Fizz" se for), múltiplo de 5 (imprimindo "Buzz" se for) ou nenhum dos anteriores (imprimindo o próprio número se for). Após cada verificação, incrementa o número.

Exercício 35: Usando um laço for, imprima os primeiros 10 números triangulares (n*(n+1)/2).

Descrição: Neste exercício, você deve imprimir os primeiros 10 números triangulares. Um número triangular é obtido através da fórmula n*(n+1)/2, onde n é a posição do número na sequência. Por exemplo, os primeiros 5 números triangulares são 1, 3, 6, 10, 15...

Código de solução:

```
for (let n = 1; n <= 10; n++) {
    let triangular = n * (n+1) / 2;
    console.log(triangular);
}
```

Explicação: Este código usa um laço for para iterar de 1 a 10. Em cada iteração, calcula o número triangular usando a fórmula n*(n+1)/2 e imprime o resultado.

Exercício 36: Usando um laço for, imprima todos os números perfeitos de 1 a 100 (um número é perfeito se a soma de seus divisores, incluindo 1 mas não ele mesmo, é igual ao próprio número).

Descrição: Neste exercício, você deve imprimir todos os números perfeitos de 1 a 100. Um número é considerado perfeito se a soma de seus divisores, incluindo 1 mas não ele mesmo, é igual ao próprio número. Por exemplo, o número 6 é perfeito, porque seus divisores são 1, 2 e 3, e 1 + 2 + 3 = 6.

Código de solução:

```
for (let i = 1; i <= 100; i++) {
    let soma = 0;
    for (let j = 1; j < i; j++) {
```

```
        if (i % j === 0) {
            soma += j;
        }
    }
    if (soma === i) {
        console.log(i);
    }
}
```

Explicação: Este código usa um laço for externo para iterar de 1 a 100. Em cada iteração, inicia uma variável soma como 0 e então usa um laço for interno para iterar de 1 até i. Se i for divisível por j, j é somado à soma. Depois que o laço interno é concluído, verifica se a soma é igual a i. Se for, imprime i como um número perfeito.

Exercício 37: Usando um laço while, imprima a soma dos quadrados dos primeiros 10 números naturais.

Descrição: Neste exercício, você deve calcular e imprimir a soma dos quadrados dos primeiros 10 números naturais. Os números naturais são os números positivos começando em 1, então os primeiros 10 números naturais são 1, 2, 3, ..., 10, e os seus quadrados são 1, 4, 9, ..., 100.

Código de solução:

```
let somaQuadrados = 0;
let i = 1;
while (i <= 10) {
    somaQuadrados += i*i;
    i++;
}
console.log(somaQuadrados);
```

Explicação: Este código inicia uma variável somaQuadrados em 0 e uma variável i em 1. Entra então em um laço while que continua até que i seja maior que 10. Em cada iteração, adiciona o quadrado de i à somaQuadrados e então incrementa i. Quando o laço termina, imprime somaQuadrados.

Exercício 38: Usando um laço do-while, imprima a soma dos cubos dos primeiros 10 números naturais.

Descrição: Neste exercício, você deve calcular e imprimir a soma dos cubos dos primeiros 10 números naturais. Os números naturais são os números positivos começando em 1, então os primeiros 10 números naturais são 1, 2, 3, ..., 10, e os seus

cubos são 1, 8, 27, ..., 1000.

Código de solução:

```
let somaCubos = 0;
let i = 1;
do {
   somaCubos += i*i*i;
   i++;
} while (i <= 10);
console.log(somaCubos);
```

Explicação: Este código inicia uma variável somaCubos em 0 e uma variável i em 1. Entra então em um laço do-while que continua até que i seja maior que 10. Em cada iteração, adiciona o cubo de i à somaCubos e então incrementa i. Quando o laço termina, imprime somaCubos.

Exercício 39: Usando um laço for, crie um programa que calcule o fatorial de um número.

Descrição: Neste exercício, você precisa criar um programa que calcula o fatorial de um número. O fatorial de um número é o produto de todos os números inteiros positivos de 1 até o número. Por exemplo, o fatorial de 5 (representado por 5!) é 1 * 2 * 3 * 4 * 5 = 120.

Código de solução:

```
let numero = 5; // Altere este valor para o número desejado.
let fatorial = 1;
for (let i = 1; i <= numero; i++) {
   fatorial *= i;
}
console.log(`O fatorial de ${numero} é ${fatorial}`);
```

Explicação: Nesse código, inicializamos a variável numero com o valor para o qual desejamos calcular o fatorial e a variável fatorial com 1. O laço for percorre todos os números de 1 até o número desejado, multiplicando o valor atual de fatorial por i. No final, o código imprime o fatorial do número.

Exercício 40: Usando um laço while, crie um programa que inverta uma string.

Descrição: Neste exercício, você precisa criar um programa que inverte uma string.

Por exemplo, a string "Hello" invertida se torna "olleH".

Código de solução:

```
let original = 'Hello'; // Altere este valor para a string desejada.
let invertida = '';
let i = original.length - 1;
while (i >= 0) {
   invertida += original[i];
   i--;
}
console.log(`A string "${original}" invertida é "${invertida}"`);
```

Explicação: Nesse código, a string original é armazenada na variável original, e a variável invertida é inicializada como uma string vazia. O laço while percorre a string original de trás para frente, adicionando cada caractere à string invertida. No final, o código imprime a string invertida.

Exercício 41: Usando um laço do-while, crie um programa que transforme um número decimal em binário.

Descrição: Neste exercício, você precisa criar um programa que transforma um número decimal em um número binário. Por exemplo, o número decimal 10 transformado em binário é 1010.

Código de solução:

```
let numero = 10; // Altere este valor para o número desejado.
let binario = '';
do {
   binario = (numero % 2) + binario;
   numero = Math.floor(numero / 2);
} while (numero > 0);
console.log(`O número em binário é ${binario}`);
```

Explicação: Nesse código, inicializamos a variável numero com o valor decimal que queremos converter para binário e a variável binario com uma string vazia. O laço do-while continua enquanto numero for maior que 0. Em cada iteração, calculamos o resto da divisão de numero por 2 e o adicionamos ao início de binario. Então, atualizamos numero para ser a parte inteira da divisão de numero por 2. No final, o código imprime o número binário.

Exercício 42: Usando um laço for, crie um programa que transforme um número binário em decimal.

Descrição: Neste exercício, você deve criar um programa que converte um número binário em um número decimal. Por exemplo, o número binário 1010 convertido em decimal é 10.

Código de solução:

```
let binario = '1010'; // Altere este valor para o número binário desejado.
let decimal = 0;
for (let i = binario.length - 1, j = 0; i >= 0; i--, j++) {
   decimal += parseInt(binario[i]) * Math.pow(2, j);
}
console.log(`O número em decimal é ${decimal}`);
```

Explicação: Neste código, a variável binario é inicializada com o número binário que queremos converter em decimal, e a variável decimal é inicializada como 0. O laço for percorre a string binária de trás para frente. Em cada iteração, convertemos o caractere binário atual em um número inteiro, multiplicamos por 2 elevado à potência do índice (que começa do 0 e vai aumentando), e somamos ao valor de decimal. No final, o código imprime o número decimal.

Exercício 43: Usando um laço while, crie um programa que determine se um número é palíndromo ou não (um número é palíndromo se for igual ao seu reverso).

Descrição: Neste exercício, você precisa criar um programa que determina se um número é um palíndromo. Um número é considerado um palíndromo se for o mesmo lido de trás para frente. Por exemplo, o número 12321 é um palíndromo.

Código de solução:

```
let numero = 12321; // Altere este valor para o número desejado.
let reverso = 0;
let temp = numero;
while (temp > 0) {
    let digito = temp % 10;
   reverso = (reverso * 10) + digito;
   temp = Math.floor(temp / 10);
}
if (numero == reverso) {
    console.log(`${numero} é um palíndromo.`);
} else {
    console.log(`${numero} não é um palíndromo.`);
}
```

Explicação: Neste código, a variável numero é inicializada com o número que queremos verificar se é um palíndromo, e a variável reverso é inicializada como 0. O laço while percorre o número digit por digit, adicionando cada digit ao reverso (mantendo a

ordem inversa). Se o número original é igual ao seu reverso, então o número é um palíndromo.

Exercício 44: Usando um laço do-while, crie um programa que imprima a soma de duas matrizes.

Descrição: Neste exercício, você deve criar um programa que soma duas matrizes. As matrizes são listas bidimensionais de números. A soma de duas matrizes é obtida somando os elementos correspondentes de cada matriz.

Código de solução:

```
let matriz1 = [[ 1 , 2 ], [ 3 , 4 ]]; // Altere estes valores para as matrizes desejadas.
let matriz2 = [[ 5 , 6 ], [ 7 , 8 ]];
let soma = [];
let i = 0 ;
do {
    let j = 0 ;
   soma[i] = [];
    do {
       soma[i][j] = matriz1[i][j] + matriz2[i][j];
       j++;
   } while (j < matriz1[i].length);
   i++;
} while (i < matriz1.length);
console .log( `A soma das matrizes é ${ JSON .stringify(soma)}` );
```

Explicação: Nesse código, inicializamos matriz1 e matriz2 com as matrizes que queremos somar, e soma com uma matriz vazia. Usamos dois laços do-while aninhados para percorrer cada elemento das matrizes. Em cada iteração, somamos os elementos correspondentes das duas matrizes e armazenamos o resultado na matriz soma. No final, o código imprime a matriz soma.

Capítulo 3: Funções em JavaScript

Funções são um dos fundamentos mais importantes de JavaScript e de qualquer outra linguagem de programação. Elas permitem que você escreva um bloco de código uma vez e o reutilize onde quer que seja necessário. Isso promove a reutilização de código, tornando seu código mais fácil de entender e de manter.

Neste capítulo, vamos explorar a poderosa funcionalidade das funções em JavaScript. As funções podem ser tão simples como uma função que adiciona dois números, ou tão complexas como uma que realiza operações de rede.

No JavaScript, funções são objetos de primeira classe, o que significa que elas podem ser passadas como argumentos para outras funções, retornadas como valores a partir de outras funções, e podem ter suas próprias propriedades e métodos.

Vamos começar com exercícios simples para entender como declarar e invocar funções. Em seguida, vamos nos aprofundar em conceitos mais avançados.

Este capítulo tem como objetivo aprofundar seu entendimento sobre funções, para que você possa usá-las efetivamente para tornar seu código mais eficiente, organizado e reutilizável.

Com os exercícios práticos deste capítulo, você terá a oportunidade de aplicar os conceitos aprendidos e aprimorar suas habilidades de programação em JavaScript.

Exercício 45: Escreva uma função que imprima "Olá, Mundo!".

Descrição: Neste exercício, você deve escrever uma função simples que não recebe argumentos e, quando chamada, imprime a frase "Olá, Mundo!".

Solução:

```
function  imprimirOlaMundo () {
    console .log( "Olá, Mundo!" );
}
imprimirOlaMundo();
```

Explicação: A função imprimirOlaMundo foi definida usando a sintaxe de função declarativa. Esta função não recebe argumentos, e seu corpo consiste apenas de uma instrução console.log que imprime a string "Olá, Mundo!". Para invocar a função, basta digitar seu nome seguido de um par de parênteses, como em imprimirOlaMundo().

Exercício 46: Escreva uma função que aceite dois números como argumentos e retorne sua soma.

Descrição: Você deve escrever uma função que receba dois números como argumentos e retorne a soma desses números.

Solução:

```
function  somarNumeros (num1, num2) {
```

```
    return  num1 + num2;
}
console .log(somarNumeros( 3 , 4 )); // Saída: 7
```

Explicação: A função somarNumeros recebe dois argumentos, num1 e num2, e retorna a soma desses dois números. Quando chamamos a função com os números 3 e 4 como argumentos, por exemplo, a função retorna a soma desses números, que é 7.

Exercício 47: Escreva uma função que aceite um número como argumento e retorne o seu quadrado.

Descrição: Neste exercício, você precisa escrever uma função que receba um número como argumento e retorne o quadrado desse número (ou seja, o número multiplicado por ele mesmo).

Solução:

```
function  quadrado (num) {
    return  num * num;
}
console .log(quadrado( 5 )); // Saída: 25
```

Explicação: A função quadrado recebe um argumento, num, e retorna o resultado da multiplicação de num por ele mesmo. Quando chamamos a função com o número 5 como argumento, por exemplo, a função retorna o quadrado desse número, que é 25.

Exercício 48: Escreva uma função que aceite um número como argumento e retorne se ele é par ou ímpar.

Descrição: Neste exercício, você precisa escrever uma função que receba um número como argumento e retorne uma string indicando se o número é par ou ímpar.

Solução:

```
function  parOuImpar (num) {
    if  (num % 2  === 0 ) {
        return  'par' ;
   } else  {
        return  'ímpar' ;
   }
}
console .log(parOuImpar( 5 )); // Saída: "ímpar"
```

Explicação: A função parOuImpar recebe um argumento, num, e retorna a string "par"

se num for par (ou seja, se num dividido por 2 tiver resto zero) e a string "ímpar" se num for ímpar (ou seja, se num dividido por 2 tiver resto diferente de zero).

Exercício 49: Escreva uma função que aceite três números como argumentos e retorne o maior deles.

Descrição: Neste exercício, você precisa escrever uma função que receba três números como argumentos e retorne o maior desses números.

Solução:

```
function maiorNumero(num1, num2, num3) {
    return Math.max(num1, num2, num3);
}
console.log(maiorNumero(3, 5, 4)); // Saída: 5
```

Explicação: A função maiorNumero recebe três argumentos, num1, num2 e num3, e retorna o maior desses três números. Para encontrar o maior número, usamos a função Math.max, que aceita qualquer número de argumentos e retorna o maior deles.

Exercício 50: Escreva uma função que aceite uma string como argumento e retorne a string invertida.

Descrição: Neste exercício, você precisa escrever uma função que receba uma string como argumento e retorne a string com seus caracteres na ordem inversa.

Solução:

```
function inverterString(str) {
    return str.split('').reverse().join('');
}
console.log(inverterString('hello')); // Saída: "olleh"
```

Explicação: A função inverterString recebe uma string str como argumento. A função split('') é usada para converter a string em um array de caracteres. O método reverse() é então usado para inverter a ordem dos elementos do array. Finalmente, o método join('') é usado para combinar os elementos do array de volta em uma string.

Exercício 51: Escreva uma função que calcule o fatorial de um número.

Descrição: O fatorial de um número é o produto de todos os números inteiros

positivos de 1 até o número. Por exemplo, o fatorial de 5 é 5*4*3*2*1 = 120. Neste exercício, você deve criar uma função que calcule o fatorial de um número.

Solução:

```
function  calcularFatorial (num) {
    let  fatorial = 1 ;
    for ( let  i = 2 ; i <= num; i++) {
       fatorial *= i;
   }
    return  fatorial;
}

console .log(calcularFatorial( 5 )); // Saída: 120
```

Explicação: A função calcularFatorial começa por inicializar uma variável fatorial com o valor 1. Em seguida, ela executa um laço for que começa em 2 e termina no número dado, inclusive. A cada passagem pelo laço, o valor de i é multiplicado ao fatorial atual. O fatorial final é então retornado.

Exercício 52: Escreva uma função que calcule o n-ésimo número de Fibonacci.

Descrição: A sequência de Fibonacci começa com os números 0 e 1, e cada número subsequente é a soma dos dois números anteriores. Neste exercício, você deve escrever uma função que retorne o n-ésimo número da sequência de Fibonacci.

Solução:

```
function  calcularFibonacci (n) {
    let  a = 0 ;
    let  b = 1 ;

    for ( let  i = 2 ; i <= n; i++) {
        let  temp = a;
       a = b;
       b = temp + b;
   }
    return  b;
}

console .log(calcularFibonacci( 7 )); // Saída: 13
```

Explicação: A função calcularFibonacci inicia duas variáveis, a e b, com os dois primeiros números da sequência de Fibonacci. Ela então executa um laço for que vai de 2 até n. A cada passagem pelo laço, o próximo número de Fibonacci é calculado pela soma de a e b, e os valores de a e b são atualizados. O número de Fibonacci solicitado é então retornado.

Exercício 53: Escreva uma função que verifique se um número é primo.

Descrição: Um número é primo se for maior do que 1 e tiver apenas dois divisores distintos: 1 e ele mesmo. Neste exercício, você deve escrever uma função que recebe um número e retorna true se o número for primo e false caso contrário.

Solução:

```
function verificarPrimo(num) {
    for (let i = 2; i < num; i++) {
        if (num % i === 0) {
            return false;
        }
    }
    return num > 1;
}
console.log(verificarPrimo(7)); // Saída: true
```

Explicação: A função verificarPrimo executa um laço for que começa em 2 e termina um número antes do dado. Se o número for divisível por i (ou seja, se num % i === 0), então o número não é primo, e a função retorna false. Se o laço termina sem encontrar um divisor, então o número é primo, e a função retorna true (assumindo que num > 1).

Exercício 54: Escreva uma função que ordene um array de números em ordem decrescente.

Descrição: Neste exercício, você deve escrever uma função que receba um array de números como argumento e retorne um novo array com os números ordenados em ordem decrescente.

Solução:

```
function ordenarDecrescente(array) {
    return array.sort((a, b) => b - a);
}
console.log(ordenarDecrescente([5, 2, 1, 3, 4])); // Saída: [5, 4, 3, 2, 1]
```

Explicação: A função ordenarDecrescente usa o método sort do array para ordenar seus elementos. Ela fornece ao método sort uma função de comparação que ordena os números em ordem decrescente. A função de comparação recebe dois números a e b e retorna a diferença entre b e a. Se b - a for positivo, b é colocado antes de a na ordenação, e vice-versa. O resultado é um novo array com os números ordenados em ordem decrescente.

Exercício 55: Escreva uma função que retorne o menor número em um array de números.

Descrição: Neste exercício, você deve criar uma função que aceita um array de números como argumento e retorna o menor número desse array.

Solução:

```
function  menorNumero (array) {
    return  Math .min(...array);
}
console .log(menorNumero([ 5 , 2 , 1 , 3 , 4 ])); // Saída: 1
```

Explicação: A função menorNumero usa a função Math.min, que retorna o menor de zero ou mais números, para encontrar o menor número do array. A sintaxe ...array é usada para passar todos os elementos do array como argumentos para a função Math.min.

Exercício 56: Escreva uma função que retorne o maior número em um array de números.

Descrição: Neste exercício, você deve criar uma função que aceita um array de números como argumento e retorna o maior número desse array.

Solução:

```
function  maiorNumero (array) {
    return  Math .max(...array);
}
console .log(maiorNumero([ 5 , 2 , 1 , 3 , 4 ])); // Saída: 5
```

Explicação: Similar à função menorNumero, a função maiorNumero usa a função Math.max, que retorna o maior de zero ou mais números, para encontrar o maior número do array. A sintaxe ...array é usada para passar todos os elementos do array como argumentos para a função Math.max.

Exercício 57: Escreva uma função que retorne a soma de todos os números ímpares em um array de números.

Descrição: Neste exercício, você deve criar uma função que aceita um array de números como argumento e retorna a soma de todos os números ímpares desse array.

Solução:

```
function somaImpares(array) {
    return array.reduce((soma, num) => num % 2 !== 0 ? soma + num : soma, 0
);
}

console.log(somaImpares([5, 2, 1, 3, 4])); // Saída: 9
```

Explicação: A função somaImpares usa o método reduce do array, que executa uma função redutora para cada elemento do array, resultando em um único valor de saída. A função redutora recebe a soma atual e o próximo número, e retorna a nova soma. Se o número é ímpar (num % 2 !== 0), ele é adicionado à soma; caso contrário, a soma permanece inalterada.

Exercício 58: Escreva uma função que retorne a soma de todos os números pares em um array de números.

Descrição: Neste exercício, você deve criar uma função que aceita um array de números como argumento e retorna a soma de todos os números pares desse array.

Solução:

```
function somaPares(array) {
    return array.reduce((soma, num) => num % 2 === 0 ? soma + num : soma, 0
);
}

console.log(somaPares([5, 2, 1, 3, 4])); // Saída: 6
```

Explicação: Similar à função somaImpares, a função somaPares usa o método reduce do array. A função redutora adiciona o número à soma se ele for par (num % 2 === 0).

Exercício 59: Escreva uma função que receba um array de números e retorne um novo array com todos os números duplicados.

Descrição: Neste exercício, você deve criar uma função que receba um array de números como argumento e retorne um novo array em que cada número seja duplicado.

Solução:

```
function duplicarNumeros(array) {
    return array.map(num => num * 2);
}

console.log(duplicarNumeros([5, 2, 1, 3, 4])); // Saída: [10, 4, 2, 6, 8]
```

Explicação: A função duplicarNumeros usa o método map do array, que cria um novo array com o resultado da chamada de uma função para cada elemento do array. A função passada para o map recebe um número e retorna o dobro desse número.

Exercício 60: Escreva uma função que receba um array de números e retorne um novo array com todos os números ao quadrado.

Descrição: Neste exercício, você deve criar uma função que receba um array de números como argumento e retorne um novo array em que cada número seja elevado ao quadrado.

Solução:

```
function quadradoNumeros(array) {
    return array.map(num => num * num);
}

console.log(quadradoNumeros([5, 2, 1, 3, 4])); // Saída: [25, 4, 1, 9, 16]
```

Explicação: Similar à função duplicarNumeros, a função quadradoNumeros usa o método map do array. A função passada para o map recebe um número e retorna o quadrado desse número.

Exercício 61: Escreva uma função que receba um array de números e retorne um novo array com a raiz quadrada de todos os números.

Descrição: Neste exercício, você deve criar uma função que receba um array de números como argumento e retorne um novo array em que cada número seja a raiz quadrada do número original.

Solução:

```
function raizQuadradaNumeros(array) {
    return array.map(num => Math.sqrt(num));
}

console.log(raizQuadradaNumeros([25, 4, 1, 9, 16])); // Saída: [5, 2, 1, 3, 4]
```

Explicação: A função raizQuadradaNumeros usa o método map do array. A função passada para o map recebe um número e retorna a raiz quadrada desse número usando a função Math.sqrt.

Exercício 62: Escreva uma função que receba um número e retorne uma string repetida aquele número de vezes.

Descrição: Neste exercício, você deve criar uma função que receba um número como argumento e retorne uma string que é repetida aquele número de vezes.

Solução:

```
function repetirString(num) {
    return 'Olá '.repeat(num);
}
console.log(repetirString(3)); // Saída: 'Olá Olá Olá'
```

Explicação: A função repetirString usa o método repeat da string, que constrói e retorna uma nova string que contém o número especificado de cópias da string original, concatenadas juntas.

Exercício 63: Escreva uma função que receba um número e retorne um array com todos os números primos até aquele número.

Descrição: Neste exercício, você deve criar uma função que receba um número como argumento e retorne um array com todos os números primos até aquele número.

Solução:

```
function primosAteN(num) {
    let primos = [];
    for (let i = 2; i <= num; i++) {
        if (isPrimo(i)) {
            primos.push(i);
        }
    }
    return primos;
}

function isPrimo(num) {
    for (let i = 2; i < num; i++) {
        if (num % i === 0) {
            return false;
        }
    }
    return num > 1;
}

console.log(primosAteN(10)); // Saída: [2, 3, 5, 7]
```

Explicação: A função primosAteN usa um loop for para verificar cada número até

num. Se o número é primo, ele é adicionado ao array primos. A função isPrimo é usada para verificar se um número é primo. Ela usa um loop for para verificar se num é divisível por qualquer número menor que ele. Se num é divisível por qualquer número, ele não é primo e isPrimo retorna false. Se num não é divisível por nenhum número, ele é primo e isPrimo retorna true.

Exercício 64: Escreva uma função que receba uma string e retorne o número de palavras na string.

Descrição: Neste exercício, você deve criar uma função que receba uma string como argumento e retorne o número de palavras na string.

Solução:

```
function contarPalavras(str) {
    return str.split(' ').length;
}
console.log(contarPalavras("Olá, Mundo!")); // Saída: 2
```

Explicação: A função contarPalavras usa o método split da string para dividir a string em um array de palavras, onde cada palavra é separada por um espaço (' '). Em seguida, usa a propriedade length do array para contar o número de palavras na string.

Exercício 65: Escreva uma função que aceite um array de números e uma função de callback e retorne a soma de todos os números do array após a aplicação da função de callback.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba um array de números e uma função de callback como argumentos, e retorne a soma de todos os números do array depois de aplicar a função de callback a cada número.

Solução:

```
function somaComCallback(array, callback) {
    return array.map(callback).reduce((a, b) => a + b, 0);
}

let numeros = [1, 2, 3, 4, 5];
let callback = num => num * 2;

console.log(somaComCallback(numeros, callback)); // Saída: 30
```

Explicação: A função somaComCallback primeiro aplica a função de callback a cada número do array usando o método map, e depois soma todos os números resultantes

usando o método reduce.

Exercício 66: Escreva uma função que aceite uma função de callback que retorne verdadeiro ou falso e um array, e retorne um novo array que contém apenas os elementos para os quais a função de callback retornou verdadeiro.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba uma função de callback e um array como argumentos, e retorne um novo array que contenha apenas os elementos para os quais a função de callback retornou verdadeiro.

Solução:

```
function filtrarPorCallback(array, callback) {
    return array.filter(callback);
}

let numeros = [1, 2, 3, 4, 5];
let callback = num => num % 2 === 0;

console.log(filtrarPorCallback(numeros, callback)); // Saída: [2, 4]
```

Explicação: A função filtrarPorCallback usa o método filter do array, que cria um novo array com todos os elementos que passam no teste implementado pela função de callback fornecida. Neste caso, a função de callback testa se um número é par.

Exercício 67: Escreva uma função que aceite uma função de callback e um array de arrays, e retorne um novo array que contém os resultados de aplicar a função de callback a cada array.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba uma função de callback e um array de arrays como argumentos, e retorne um novo array que contenha os resultados de aplicar a função de callback a cada array.

Solução:

```
function mapearArrays(array, callback) {
    return array.map(callback);
}

let arrays = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
let callback = arr => arr.reduce((a, b) => a + b, 0);

console.log(mapearArrays(arrays, callback)); // Saída: [6, 15, 24]
```

Explicação: A função mapearArrays usa o método map do array, que cria um novo array com os resultados de chamar uma função de callback fornecida em cada elemento do array. Neste caso, a função de callback soma todos os elementos de cada array.

Exercício 68: Escreva uma função de alta ordem que aceite um array de números e uma função de callback, e retorne um novo array que contém apenas os números do array original que satisfazem a condição da função de callback.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba um array de números e uma função de callback como argumentos, e retorne um novo array que contenha apenas os números do array original que passam no teste implementado pela função de callback.

Solução:

```
function filtrarNumeros(array, callback) {
    return array.filter(callback);
}

let numeros = [1, 2, 3, 4, 5];
let callback = num => num > 3;

console.log(filtrarNumeros(numeros, callback)); // Saída: [4, 5]
```

Explicação: A função filtrarNumeros usa o método filter do array, que cria um novo array com todos os elementos que passam no teste implementado pela função de callback fornecida. Neste caso, a função de callback testa se um número é maior que 3.

Exercício 69: Escreva uma função que aceite um número e uma função de callback, e execute a função de callback o número de vezes especificado.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba um número e uma função de callback como argumentos, e execute a função de callback o número de vezes especificado.

Solução:

```
function executarCallback(n, callback) {
    for (let i = 0; i < n; i++) {
       callback(i);
   }
}

let callback = i => console.log(`Execução ${i+1}`);
```

```
executarCallback( 5 , callback); // Saída: "Execução 1", "Execução 2",
"Execução 3", "Execução 4", "Execução 5"
```

Explicação: A função executarCallback usa um loop for para executar a função de callback n vezes. A função de callback recebe o número da execução atual como argumento e imprime uma mensagem.

Exercício 70: Escreva uma função de alta ordem que aceite um array de funções de callback e um valor, e retorne um array de resultados de aplicar cada função de callback ao valor.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba um array de funções de callback e um valor como argumentos, e retorne um array que contenha os resultados de aplicar cada função de callback ao valor.

Solução:

```
function  aplicarCallbacks (callback, valor) {
    return  callback.map(callback => callback(valor));
}

let  callback = [num => num * 2 , num => num * num, num => num / 2 ];
let  valor = 4 ;

console .log(aplicarCallbacks(callback, valor)); // Saída: [8, 16, 2]
```

Explicação: A função aplicarCallbacks usa o método map do array, que cria um novo array com os resultados de chamar uma função de callback fornecida em cada elemento do array. Neste caso, cada função de callback é aplicada ao valor fornecido.

Exercício 71: Escreva uma função que aceite uma função de callback e um número, e execute a função de callback após um certo número de milissegundos especificados pelo número.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba uma função de callback e um número como argumentos, e execute a função de callback após um certo número de milissegundos especificado pelo número. Isso é conhecido como programação assíncrona.

Solução:

```
function  executarDepois (callback, tempo) {
   setTimeout(callback, tempo);
}
```

```
let  callbaack = () => console .log( "Executado!" );

executarDepois(callbaack, 2000 ); // Saída: "Executado!" após 2 segundos
```

Explicação: A função executarDepois usa a função setTimeout do JavaScript, que executa uma função de callback após um certo número de milissegundos. O primeiro argumento para setTimeout é a função de callback a ser executada, e o segundo argumento é o número de milissegundos antes da execução da função de callback.

Exercício 72: Escreva uma função que aceite um número, um array de funções de callback e um valor inicial, e retorne o resultado final após a aplicação de todas as funções de callback no valor inicial, o número de vezes especificado.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba um número, um array de funções de callback e um valor inicial como argumentos, e retorne o resultado final após a aplicação de todas as funções de callback no valor inicial, o número de vezes especificado.

Solução:

```
function  aplicarCallbacksNTimes (n, callback, valorInicial) {
    let  valor = valorInicial;
    for ( let  i = 0 ; i < n; i++){
       callback.forEach(callback => {
           valor = callback(valor);
       });
   }
    return  valor;
}

let  callback = [num => num * 2 , num => num + 1 ];
let  valorInicial = 1 ;

console .log(aplicarCallbacksNTimes( 2 , callback, valorInicial)); // Saída: 7
```

Explicação: A função aplicarCallbacksNTimes usa um loop for para aplicar cada função de callback no valor atual n vezes. O valor resultante de uma função de callback é usado como entrada para a próxima função de callback. O valor resultante final é retornado.

Exercício 73: Escreva uma função de alta ordem que aceite um número, um array de funções de callback e um valor inicial, e retorne o resultado final após a aplicação de todas as funções de callback no valor inicial, a cada "n"

milissegundos.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba um número, um array de funções de callback e um valor inicial como argumentos, e retorne o resultado final após a aplicação de todas as funções de callback no valor inicial, a cada "n" milissegundos. Isso é conhecido como programação assíncrona.

Solução:

```
function  aplicarCallbacksComIntervalo (n, callback, valorInicial) {
    let  valor = valorInicial;
    let  i = 0 ;
    let  intervalId = setInterval(() => {
        if (i < callback.length) {
          valor = callback[i](valor);
       i++;
       } else {
          clearInterval(intervalId);
          console .log(valor);
       }
   }, n);
}

let  callback = [num => num * 2 , num => num + 1 ];
let  valorInicial = 1 ;

aplicarCallbacksComIntervalo( 2000 , callback, valorInicial); // Saída: valor
após 2 segundos por callback
```

Explicação: A função aplicarCallbacksComIntervalo usa a função setInterval do JavaScript, que executa uma função de callback a cada "n" milissegundos. Neste caso, cada função de callback é aplicada ao valor atual a cada "n" milissegundos. O valor resultante de uma função de callback é usado como entrada para a próxima função de callback. A execução é interrompida após n vezes usando a função clearInterval.

Exercício 74: Escreva uma função de alta ordem que aceite uma função de callback e um array, e retorne um novo array que contém os resultados de aplicar a função de callback a cada elemento do array, sem alterar o array original.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba uma função de callback e um array como argumentos, e retorne um novo array que contenha os resultados de aplicar a função de callback a cada elemento do array. A função de callback deve ser uma função pura, ou seja, não deve alterar o estado do array original.

Solução:

```
function  mapearSemAlterar (array, callback) {
    return  array.map(callback);
}

let  numeros = [ 1 , 2 , 3 , 4 , 5 ];
let  callback = num => num * 2 ;

console .log(mapearSemAlterar(numeros, callback)); // Saída: [2, 4, 6, 8, 10]
console .log(numeros); // Saída: [1, 2, 3, 4, 5]
```

Explicação: A função mapearSemAlterar usa o método map do array, que cria um novo array com os resultados de chamar uma função de callback fornecida em cada elemento do array. Como a função de callback é uma função pura, ela não altera o estado do array original. Neste caso, a função de callback duplica cada número.

Exercício 75: Escreva uma função de alta ordem que aceite um array de funções de callback e um valor, e retorne um novo array que contém os resultados de aplicar a função de callback ao valor, na ordem inversa.

Descrição: Neste exercício, você deve criar uma função de alta ordem que receba um array de funções de callback e um valor como argumentos, e retorne um novo array que contenha os resultados de aplicar cada função de callback ao valor, na ordem inversa.

Solução:

```
function  aplicarCallbacksInversamente (callback, valor) {
    return  callback.reverse().map(callback => callback(valor));
}

let  callback = [num => num * 2 , num => num * num, num => num / 2 ];
let  valor = 4 ;

console .log(aplicarCallbacksInversamente(callback, valor)); // Saída: [2,
16, 8]
```

Explicação: A função aplicarCallbacksInversamente primeiro reverte a ordem das funções de callback usando o método reverse do array, e então usa o método map do array para criar um novo array com os resultados de chamar uma função de callback fornecida em cada elemento do array. Neste caso, cada função de callback é aplicada ao valor fornecido.

Capítulo 4: Objetos e Arrays

Neste capítulo, trabalharemos profundamente na manipulação de Objetos e Arrays em JavaScript.

Esses dois tipos de dados são essenciais para a estruturação e manipulação de dados em quase todos os programas JavaScript.
Vamos explorar como criar, acessar, modificar e iterar sobre Objetos e Arrays, assim como entender a importância da mutabilidade e referência na manipulação dessas estruturas de dados.

Mas antes de começarmos, deixo aqui a sugestão do curso de Orientação a Objetos com JavaScript , tema que vamos abordar neste capítulo.

Objetos: Propriedades, Métodos e Herança

Os Objetos são uma das estruturas de dados mais fundamentais em JavaScript, e uma compreensão profunda de como eles funcionam é essencial para ser um bom programador JavaScript.

Nesta seção, começaremos explorando o básico dos objetos - como definir e acessar propriedades e métodos.

Então, entraremos em tópicos mais avançados como a herança de objetos, a cadeia de protótipos e a diferença entre funções construtoras, classes e objetos literais.

Essas ferramentas e conceitos nos permitirão escrever código mais reutilizável e eficiente.

Exercício 76: Crie um objeto vazio e adicione três propriedades a ele, cada uma com diferentes tipos de valores.

Descrição: O JavaScript permite que objetos sejam criados e modificados dinamicamente. Neste exercício, crie um objeto vazio. Em seguida, adicione uma propriedade numérica, uma propriedade string e uma propriedade booleana.

Solução:

```
let  obj = {}; // Criação de um objeto vazio

obj.numero = 10 ; // Adicionando uma propriedade numérica
obj.texto = "Olá, mundo!" ; // Adicionando uma propriedade string
obj.booleano = true ; // Adicionando uma propriedade booleana

console .log(obj); // Deve imprimir { numero: 10, texto: 'Olá, mundo!',
booleano: true }
```

Explicação: Nesse exercício, um objeto vazio foi criado com a sintaxe let obj = {};. Em seguida, foram adicionadas três propriedades com diferentes tipos de valores. Para adicionar uma propriedade a um objeto em JavaScript, basta utilizar a sintaxe

obj.nomeDaPropriedade = valor;. Ao final, imprimimos o objeto para visualizarmos as propriedades adicionadas.

Exercício 77: Crie um objeto que represente um livro, com propriedades para o título, autor e número de páginas. Em seguida, adicione um método ao objeto que imprima uma descrição do livro.

Descrição: Neste exercício, você vai criar um objeto que represente um livro. O objeto deve ter propriedades para o título, autor e número de páginas. Além disso, o objeto deve ter um método que imprima uma descrição do livro.

Solução:

```
let  livro = {
 titulo: 'O Hobbit' ,
 autor: 'J.R.R. Tolkien' ,
 numeroDePaginas: 310 ,


 descricao: function () {
    console .log( `O livro ${ this .titulo}, escrito por ${ this .autor}, tem
${ this .numeroDePaginas} páginas.` );
 }
};

livro.descricao(); // Deve imprimir: 'O livro O Hobbit, escrito por J.R.R.
Tolkien, tem 310 páginas.'
```

Explicação: Neste exercício, foi criado um objeto livro com propriedades e um método. As propriedades são definidas com a sintaxe propriedade: valor,, e o método é uma função que é atribuída a uma propriedade. Note que no método descricao, utilizamos a palavra-chave this para se referir ao próprio objeto livro.

Exercício 78: Crie um objeto "cachorro" com propriedades para raça, nome e idade. Adicione um método que retorne a idade do cachorro em anos humanos (idade do cachorro * 7).

Descrição: Neste exercício, você precisa criar um objeto que represente um cachorro. Este objeto deve ter propriedades para raça, nome e idade. Em seguida, adicione um método que converta a idade do cachorro em anos humanos (multiplique a idade do cachorro por 7).

Solução:

```
let  cachorro = {
 raca: 'Labrador' ,
 nome: 'Rex' ,
 idade: 3 ,

 idadeEmAnosHumanos: function () {
    return  this .idade * 7 ;
 }
};
console .log(cachorro.idadeEmAnosHumanos()); // Deve imprimir: 21
```

Explicação: Neste exercício, foi criado um objeto cachorro com três propriedades: raca, nome e idade. Além disso, foi adicionado um método idadeEmAnosHumanos que retorna a idade do cachorro em anos humanos (a idade do cachorro multiplicada por 7).

Exercício 79: Crie um objeto que represente um carro, com propriedades para a marca, modelo, ano e velocidade atual, e os seguintes métodos: acelerar, frear e obter velocidade atual.

Descrição: Neste exercício, você deve criar um objeto que represente um carro. O carro deve ter propriedades para marca, modelo, ano e velocidade atual. Em seguida, adicione os seguintes métodos ao objeto: um método que aumenta a velocidade atual, um método que diminui a velocidade atual e um método que retorna a velocidade atual.

Solução:

```
let  carro = {
 marca: 'Ford' ,
 modelo: 'Mustang' ,
 ano: 1969 ,
 velocidadeAtual: 0 ,

 acelerar: function () {
    this .velocidadeAtual += 10 ;
 },

 frear: function () {
    this .velocidadeAtual -= 10 ;
    if  ( this .velocidadeAtual < 0 ) {
      this .velocidadeAtual = 0 ;
   }
 },

 obterVelocidadeAtual: function () {
```

```
    return  this.velocidadeAtual;
 }
};

carro.acelerar();
console.log(carro.obterVelocidadeAtual()); // Deve imprimir: 10

carro.frear();
console.log(carro.obterVelocidadeAtual()); // Deve imprimir: 0
```

Explicação: Neste exercício, foi criado um objeto carro com quatro propriedades e três métodos. As propriedades são marca, modelo, ano e velocidadeAtual. Os métodos são acelerar, frear e obterVelocidadeAtual. O método acelerar aumenta a velocidadeAtual em 10 unidades. O método frear diminui a velocidadeAtual em 10 unidades, mas garante que a velocidadeAtual nunca seja menor que 0. O método obterVelocidadeAtual simplesmente retorna a velocidadeAtual.

Exercício 80: Crie um objeto que represente um estudante, com propriedades para o nome, notas de várias matérias e um método para calcular a média das notas.

Descrição: Neste exercício, você deve criar um objeto que represente um estudante. O objeto estudante deve ter propriedades para o nome do estudante e as notas que ele tirou em várias matérias. Em seguida, adicione um método ao objeto que calcule a média das notas do estudante.

Solução:

```
let estudante = {
 nome: 'João',
 notas: [10, 8, 9, 7],

 calcularMedia: function() {
    let soma = 0;
    for(let i = 0; i < this.notas.length; i++) {
     soma += this.notas[i];
   }
    return soma / this.notas.length;
 }
};

console.log(estudante.calcularMedia()); // Deve imprimir: 8.5
```

Explicação: Neste exercício, foi criado um objeto estudante com duas propriedades: nome e notas. A propriedade notas é um array de números. Além disso, foi adicionado um método calcularMedia que retorna a média das notas do estudante. Para calcular a média, somamos todas as notas e dividimos pelo número de notas.

Exercício 81: Crie um objeto representando uma conta bancária, que possui uma propriedade de saldo e métodos para depósito e saque.

Descrição: Neste exercício, você precisa criar um objeto que represente uma conta bancária. Este objeto deve ter uma propriedade para o saldo e métodos para realizar um depósito e um saque.

Solução:

```
let  contaBancaria = {
 saldo: 1000 ,


 depositar: function (valor) {
    this .saldo += valor;
 },


 sacar: function (valor) {
    if (valor <= this .saldo) {
      this .saldo -= valor;
   } else  {
      console .log( 'Saldo insuficiente.' );
   }
 },


 obterSaldo: function () {
    return  this .saldo;
 }
};

contaBancaria.depositar( 500 );
console .log(contaBancaria.obterSaldo()); // Deve imprimir: 1500

contaBancaria.sacar( 200 );
console .log(contaBancaria.obterSaldo()); // Deve imprimir: 1300
```

Explicação: Neste exercício, foi criado um objeto contaBancaria com uma propriedade saldo e três métodos: depositar, sacar e obterSaldo. O método depositar acrescenta um valor ao saldo. O método sacar retira um valor do saldo, mas verifica se há saldo suficiente antes de realizar o saque. O método obterSaldo retorna o saldo atual.

Exercício 82: Crie um objeto "circulo" que possui uma propriedade de raio e dois métodos que calculam a área e a circunferência.

Descrição: Neste exercício, você precisa criar um objeto que represente um círculo. O

círculo deve ter uma propriedade para o raio e dois métodos, um para calcular a área do círculo (pi * raio^2) e outro para calcular a circunferência do círculo (2 * pi * raio).

Solução:

```
let circulo = {
 raio: 5,

 calcularArea: function() {
    return Math.PI * Math.pow(this.raio, 2);
 },

 calcularCircunferencia: function() {
    return 2 * Math.PI * this.raio;
 }
};

console.log(circulo.calcularArea()); // Deve imprimir: 78.53981633974483
console.log(circulo.calcularCircunferencia()); // Deve imprimir:
31.41592653589793
```

Explicação: Neste exercício, foi criado um objeto circulo com uma propriedade raio e dois métodos: calcularArea e calcularCircunferencia. O método calcularArea retorna a área do círculo, calculada como pi * raio^2. O método calcularCircunferencia retorna a circunferência do círculo, calculada como 2 * pi * raio. Utilizamos o valor de pi disponível na biblioteca Math do JavaScript.

Exercício 83: Crie um objeto "tempo" que possui propriedades para horas, minutos e segundos, e um método para converter o tempo para segundos.

Descrição: Neste exercício, você precisa criar um objeto que represente um tempo. O tempo deve ter propriedades para horas, minutos e segundos. Em seguida, adicione um método ao objeto que converta o tempo para segundos.

Solução:

```
let tempo = {
 horas: 2,
 minutos: 30,
 segundos: 15,

 converterParaSegundos: function() {
    return this.horas * 3600 + this.minutos * 60 + this.segundos;
 }
};

console.log(tempo.converterParaSegundos()); // Deve imprimir: 9015
```

Explicação: Neste exercício, foi criado um objeto tempo com três propriedades: horas, minutos e segundos. Além disso, foi adicionado um método converterParaSegundos que retorna o tempo total em segundos. Para isso, multiplicamos as horas por 3600 (pois uma hora tem 3600 segundos), os minutos por 60 (pois um minuto tem 60 segundos) e somamos com os segundos.

Exercício 84: Crie um objeto "retangulo" que possui propriedades para altura e largura, e um método para calcular a área.

Descrição: Neste exercício, você precisa criar um objeto que represente um retângulo. O retângulo deve ter propriedades para a altura e a largura. Em seguida, adicione um método ao objeto que calcule a área do retângulo.

Solução:

```
let  retangulo = {
 altura: 5 ,
 largura: 10 ,


 calcularArea: function () {
    return  this .altura * this .largura;
 }
};
console .log(retangulo.calcularArea()); // Deve imprimir: 50
```

Explicação: Neste exercício, foi criado um objeto retangulo com duas propriedades: altura e largura. Além disso, foi adicionado um método calcularArea que retorna a área do retângulo, que é calculada multiplicando a altura pela largura.

Exercício 85: Crie um objeto "quadrado" que herda as propriedades do objeto "retangulo" e substitua o método para calcular a área.

Descrição: Neste exercício, você precisa criar um objeto que represente um quadrado. Este objeto deve herdar as propriedades do objeto retangulo criado no exercício anterior. Em seguida, substitua o método para calcular a área para que utilize apenas uma das dimensões (já que a altura e a largura de um quadrado são iguais).

Solução:

```
let  quadrado = Object .create(retangulo);
quadrado.largura = 5 ;
quadrado.altura = 5 ;
```

```
quadrado.calcularArea = function () {
  return  this .largura * this .largura;
};

console .log(quadrado.calcularArea()); // Deve imprimir: 25
```

Explicação: Neste exercício, foi criado um objeto quadrado que herda as propriedades do objeto retangulo usando a função Object.create(). Em seguida, as propriedades largura e altura do quadrado foram ajustadas para 5 (já que em um quadrado a largura e a altura são iguais). Por fim, foi substituído o método calcularArea para que retorne a área do quadrado, que é calculada multiplicando a largura pela largura (ou a altura pela altura).

Exercício 86: Crie um objeto "livro" que possui propriedades para o título, autor e número de páginas, e um método para exibir o livro na console.

Descrição: Neste exercício, você precisa criar um objeto que represente um livro. O livro deve ter propriedades para o título, autor e número de páginas. Em seguida, adicione um método ao objeto que exiba o livro na console no formato "Título, escrito por Autor, com Páginas páginas".

Solução:

```
let  livro = {
 titulo: '1984' ,
 autor: 'George Orwell' ,
 paginas: 328 ,


 exibirLivro: function () {
    console .log( this .titulo + ', escrito por '  + this .autor + ', com '  +
this .paginas + ' páginas' );
 }
};

livro.exibirLivro(); // Deve imprimir: 1984, escrito por George Orwell, com
328 páginas
```

Explicação: Neste exercício, foi criado um objeto livro com três propriedades: titulo, autor e paginas. Além disso, foi adicionado um método exibirLivro que imprime o livro na console no formato especificado no enunciado do exercício.

Arrays: Métodos e Operações

Os arrays, também conhecidos como vetores ou listas, são uma estrutura de dados muito importante em qualquer linguagem de programação, incluindo JavaScript. Um array é uma coleção ordenada de elementos que podem ser de qualquer tipo: números, strings, objetos, outros arrays e até mesmo funções.

Em JavaScript, além de simplesmente armazenar e recuperar dados, os arrays têm uma série de métodos úteis para manipulá-los. Esses métodos permitem realizar operações como adicionar ou remover elementos do array, percorrer e modificar seus elementos, pesquisar por elementos específicos, ordenar os elementos, entre outras operações.

Nesta seção, vamos aprender a utilizar esses métodos e operações em arrays. Para cada um dos exercícios, você verá um exemplo prático de como usar o método ou operação em questão, juntamente com uma explicação detalhada. Através desses exercícios, você ganhará uma compreensão mais profunda do poder e da flexibilidade dos arrays em JavaScript.

Exercício 87: Dado dois arrays, um com vários números pares e outro com números ímpares, combine-os e, em seguida, filtre apenas os números que são múltiplos de 5.

Descrição:

Você deve criar uma função que aceite dois arrays como argumento e retorne um novo array resultante da combinação dos dois, mas filtrado para conter apenas múltiplos de 5.

Solução:

```
function  combineAndFilter (pares, impares) {
    return  pares.concat(impares).filter(num => num % 5  === 0 );
}

const  pares = [ 2 , 4 , 10 , 12 , 20 ];
const  impares = [ 1 , 3 , 5 , 15 , 25 ];
console .log(combineAndFilter(pares, impares)); // [10, 20, 5, 15, 25]
```

Explicação:

Usamos o método concat() para combinar os dois arrays e o método filter() para filtrar apenas os números múltiplos de 5.

Exercício 88: Dado um array de preços de produtos, calcule o preço total após adicionar 10% de imposto a cada produto usando map(), e depois somando tudo com reduce().

Descrição:

Você deve criar uma função que aceite um array de números como argumento e retorne o valor total depois de calcular o imposto.

Solução:

```
function calculateTotalWithTax(prices) {
    return prices.map(price => price * 1.10).reduce((acc, curr) => acc + curr, 0);
}

const prices = [10, 20, 30, 40];
console.log(calculateTotalWithTax(prices)); // 110
```

Explicação:

Usamos o método map() para adicionar 10% a cada preço e o método reduce() para somar os valores atualizados.

Exercício 89: Dada uma string com várias palavras separadas por espaços, transforme-a em um array e depois retorne a palavra mais longa.

Descrição:

Você deve criar uma função que aceite uma string e retorne a palavra mais longa encontrada.

Solução:

```
function longestWord(string) {
    return string.split(" ").reduce((acc, curr) => curr.length > acc.length ? curr : acc, "");
}

const sentence = "Essa frase tem muitas palavras, qual é a maior?";
console.log(longestWord(sentence)); // "palavras"
```

Explicação:

Usamos o método split() para transformar a string em um array e reduce() para encontrar a palavra mais longa.

Exercício 90: Dado um array que contém vários elementos duplicados, crie uma função que remova todas as duplicatas e retorne um array com elementos únicos.

Descrição:

Você deve criar uma função que aceite um array e retorne um novo array sem elementos duplicados.

Solução:

```
function removeDuplicates(arr) {
    return [...new Set(arr)];
}

const numbers = [1, 2, 3, 2, 4, 3, 5, 6, 7, 5, 8];
console.log(removeDuplicates(numbers)); // [1, 2, 3, 4, 5, 6, 7, 8]
```

Explicação:

O Set é uma estrutura de dados que armazena valores únicos. Ao passar o array para um Set e então espalhá-lo de volta para um array, removemos todos os duplicados.

Exercício 91: Dado um array de strings, ordene-o em ordem decrescente com base no comprimento de cada string.

Descrição:

Você deve criar uma função que aceite um array de strings e retorne um novo array ordenado pelo comprimento das strings.

Solução:

```
function sortByStringLength(arr) {
    return arr.sort((a, b) => b.length - a.length);
}

const words = ["apple", "banana", "cherry", "date", "elderberry"];
console.log(sortByStringLength(words)); // ["elderberry", "banana",
"cherry", "apple", "date"]
```

Explicação:

O método sort() foi utilizado com uma função de comparação para ordenar o array com base no comprimento das strings.

Exercício 92: Dado um array de objetos que têm várias propriedades, crie uma função que retorne um novo array contendo apenas os valores de uma propriedade específica.

Descrição:

Você deve criar uma função que aceite um array de objetos e uma chave de propriedade, e retorne um novo array contendo os valores dessa propriedade.

Solução:

```
function extractPropertyValues(arr, key) {
    return arr.map(obj => obj[key]);
}

const users = [
   {name: "John", age: 25},
   {name: "Jane", age: 30},
   {name: "Doe", age: 20}
];
console.log(extractPropertyValues(users, "name")); // ["John", "Jane",
"Doe"]
```

Explicação:

Utilizamos o método map() para extrair os valores da propriedade especificada de cada objeto no array.

Exercício 93: Dado um array de números, encontre a mediana.

Descrição:

Você deve criar uma função que aceite um array de números e retorne a mediana.

Solução:

```
function findMedian(arr) {
   arr.sort((a, b) => a - b);
    const middle = Math.floor(arr.length / 2);
    if (arr.length % 2 === 0) {
        return (arr[middle - 1] + arr[middle]) / 2;
   }
    return arr[middle];
}

const numbers = [5, 2, 9, 1, 5, 6];
console.log(findMedian(numbers)); // 5
```

Explicação:

Primeiro, ordenamos o array. Se o comprimento do array for par, a mediana é a média dos dois números do meio. Se for ímpar, a mediana é o número do meio.

Exercício 94: Dado um array de strings e um número n, retorne um novo array contendo apenas as strings que têm mais de n caracteres.

Descrição:

Você deve criar uma função que aceite um array de strings e um número, e retorne um novo array filtrado.

Solução:

```
function filterByStringLength(arr, n) {
    return arr.filter(word => word.length > n);
}

const words = ["apple", "banana", "cherry", "date", "elderberry"];
console.log(filterByStringLength(words, 5)); // ["banana", "elderberry"]
```

Explicação:

Usamos o método filter() para filtrar as strings cujo comprimento é maior que n.

Exercício 95: Dado dois arrays, retorne um novo array que é a concatenação dos dois.

Descrição:
Você deve criar uma função que aceite dois arrays e retorne um novo array que é a concatenação de ambos.

Solução:

```
function concatenateArrays(arr1, arr2) {
    return arr1.concat(arr2);
}

const firstArray = [1, 2, 3];
const secondArray = [4, 5, 6];

console.log(concatenateArrays(firstArray, secondArray)); // [1, 2, 3, 4, 5, 6]
```

Explicação:

O método concat() é usado para juntar dois ou mais arrays.

Exercício 96: Dado dois arrays, verifique se o segundo array é um subarray do primeiro.

Descrição:

Você deve criar uma função que aceite dois arrays e retorne true se o segundo array for um subarray do primeiro, e false caso contrário.

Solução:

```
function  isSubarray (mainArray, subArray) {
    return  mainArray.join( ',' ).includes(subArray.join( ',' ));
}

const  mainArr = [ 1 , 2 , 3 , 4 , 5 ];
const  subArr1 = [ 2 , 3 , 4 ];
const  subArr2 = [ 2 , 4 , 5 ];
console .log(isSubarray(mainArr, subArr1)); // true
console .log(isSubarray(mainArr, subArr2)); // false
```

Explicação:

Convertendo ambos os arrays para strings usando join() e depois verificando a inclusão usando includes().

Exercício 97: Dado um array, encontre o elemento que aparece mais vezes.

Descrição:

Você deve criar uma função que aceite um array e retorne o elemento mais frequente.

Solução:

```
function  mostFrequentItem (arr) {
    const  frequency = {};
    let  maxCount = 0 ;
    let  mostFrequent;

    for  ( let  item of  arr) {
        if  (frequency[item]) {
            frequency[item]++;
        } else  {
            frequency[item] = 1 ;
        }

        if  (frequency[item] > maxCount) {
            maxCount = frequency[item];
            mostFrequent = item;
        }
    }

    return  mostFrequent;
}
```

```
const numbers = [1, 2, 3, 2, 4, 3, 5, 2];
console.log(mostFrequentItem(numbers)); // 2
```

Explicação:

A função usa um objeto para rastrear a frequência de cada elemento. Em seguida, identifica o item com a contagem máxima.

Exercício 98: Dado um array de strings e um array de palavras proibidas, retorne um novo array sem as palavras proibidas.

Descrição:

Você deve criar uma função que aceite um array de strings e um array de palavras proibidas, e retorne um novo array filtrado.

Solução:

```
function filterOutWords(words, forbiddenWords) {
    return words.filter(word => !forbiddenWords.includes(word));
}

const allWords = ["apple", "banana", "cherry", "date"];
const forbidden = ["banana", "date"];
console.log(filterOutWords(allWords, forbidden)); // ["apple", "cherry"]
```

Explicação:

Usamos o método filter() para excluir palavras que estão presentes no array de palavras proibidas.

Exercício 99: Dado um array, retorne um novo array com todas as combinações possíveis de pares de elementos.

Descrição:

Você deve criar uma função que aceite um array e retorne outro array com todos os pares possíveis de elementos.

Solução:

```
function combineInPairs(arr) {
    const result = [];
    for (let i = 0; i < arr.length; i++) {
        for (let j = i + 1; j < arr.length; j++) {
            result.push([arr[i], arr[j]]);
```

```
        }
    }
     return result;
}

const numbers = [1, 2, 3, 4];
console.log(combineInPairs(numbers)); // [[1,2], [1,3], [1,4], [2,3], [2,4],
[3,4]]
```

Explicação:

Utilizamos dois loops for para criar todas as combinações possíveis de pares.

Exercício 100: Dado um array e um número n, divida o array em várias sub-arrays onde cada sub-array tem no máximo n elementos.

Descrição:

Você deve criar uma função que aceite um array e um número, e retorne um novo array composto de sub-arrays.

Solução:

```
function chunkArray(arr, n) {
    const chunks = [];
    for (let i = 0; i < arr.length; i += n) {
       chunks.push(arr.slice(i, i + n));
   }
    return chunks;
}

const items = [1, 2, 3, 4, 5, 6, 7];
console.log(chunkArray(items, 3)); // [[1,2,3], [4,5,6], [7]]
```

Explicação:

Utilizamos um loop for para dividir o array original em sub-arrays de tamanho n usando o método slice().

Exercício 101: Dado um array de números, retorne a soma de todos os valores únicos.

Descrição:

Você deve criar uma função que aceite um array de números e retorne a soma dos números únicos.

Solução:

```
function sumUniqueValues(arr) {
    return arr.filter(num => arr.indexOf(num) ===
arr.lastIndexOf(num)).reduce((acc, curr) => acc + curr, 0);
}

const numbers2 = [1, 2, 3, 2, 3, 4, 5];
console.log(sumUniqueValues(numbers2)); // 1 + 4 + 5 = 10
```

Explicação:

Usamos filter() para pegar números que aparecem apenas uma vez no array. Em seguida, usamos reduce() para somar esses valores únicos.

Exercício 102: Dado um array de strings, retorne uma única string que é a concatenação de todas as strings do array, separadas por vírgulas.

Descrição:

Você deve criar uma função que aceite um array de strings e retorne uma única string.

Solução:

```
function joinStrings(arr) {
    return arr.join(",");
}

const fruits = ["apple", "banana", "cherry"];
console.log(joinStrings(fruits)); // "apple,banana,cherry"
```

Explicação:

O método join() é usado para juntar todos os elementos de um array em uma string, separados pelo delimitador fornecido.

JSON:

JavaScript Object Notation, conhecido por suas siglas JSON, tornou-se um dos formatos de dados mais amplamente adotados na web. Em sua essência, o JSON é um formato leve para intercâmbio de dados. Por ser um subconjunto da linguagem JavaScript, ele é facilmente interpretado e gerado por muitas linguagens de programação modernas, o que contribui para sua popularidade.

O JSON é frequentemente usado para transmitir dados entre um servidor e um cliente

na web, ou para armazenar configurações e informações em aplicativos. Por ser simples e fácil de ler, muitos desenvolvedores preferem o JSON em detrimento de outros formatos de intercâmbio de dados, como o XML.

Nesta seção, exploraremos as principais operações e manipulações relacionadas ao JSON em JavaScript, ajudando a aprimorar sua capacidade de lidar com dados neste formato. Desde a interpretação e criação de strings JSON até o manuseio de suas estruturas internas, estes exercícios fornecerão as habilidades práticas necessárias para operar com JSON de forma eficiente.

Exercício 103: Convertendo Objetos em Strings JSON

Descrição:
Utilize o método JSON.stringify() para converter um objeto em uma string JSON. Por exemplo, o objeto { nome: "João", idade: 25 } deve ser convertido para a string '{"nome":"João","idade":25}'.

Solução:

```
function objetoParaJson(obj) {
    return JSON.stringify(obj);
}

const objetoTeste = { nome: "João", idade: 25 };
console.log(objetoParaJson(objetoTeste)); // "{'nome':'João','idade':25}"
```

Explicação:

A função objetoParaJson recebe um objeto como argumento e utiliza a função JSON.stringify() para convertê-lo em uma string JSON. O resultado é então retornado.

Exercício 104: Convertendo Strings JSON em Objetos

Descrição:

Utilize o método JSON.parse() para converter uma string em um objeto. Por exemplo, a string '{"nome":"João","idade":25}' deve ser convertida para o objeto { nome: "João", idade: 25 }.

Solução:

```
function jsonParaObjeto(jsonStr) {
    return JSON.parse(jsonStr);
}

const jsonStringTeste = '{"nome":"João","idade":25}';
```

```
console.log(jsonParaObjeto(jsonStringTeste));
// {
//   "nome": "João",
//   "idade": 25
// }
```

Explicação:
A função jsonParaObjeto recebe uma string JSON como argumento e utiliza a função JSON.parse() para convertê-la em um objeto JavaScript. O objeto resultante é então retornado.

Exercício 105: Acessando Propriedades de um Objeto Convertido

Descrição:

Dado uma string JSON que representa um objeto, converta essa string e acesse uma propriedade específica do objeto convertido.

Solução:

```
function acessarPropriedade(jsonStr, prop) {
    const obj = JSON.parse(jsonStr);
    return obj[prop];
}

const jsonStringTeste = '{"nome":"João","idade":25}';
console.log(acessarPropriedade(jsonStringTeste, "nome")); // "João"
```

Explicação:

A função acessarPropriedade primeiro converte a string JSON em um objeto JavaScript e, em seguida, acessa a propriedade desejada do objeto, retornando o valor da propriedade.

Exercício 106: Adicionando Propriedades a um Objeto Convertido

Descrição:

Depois de converter a string usando JSON.parse(), adicione uma nova propriedade ao objeto e retorne o objeto atualizado.

Solução:

```
function adicionarPropriedade(jsonStr, chave, valor) {
    const obj = JSON.parse(jsonStr);
   obj[chave] = valor;
```

```
    return obj;
}

const jsonStringTeste = '{"nome":"João","idade":25}';
console.log(adicionarPropriedade(jsonStringTeste, "profissao", "Engenheiro"
));
// {
//   "nome": "João",
//   "idade": 25,
//   "profissao": "Engenheiro"
// }
```

Explicação:

A função adicionarPropriedade converte a string JSON em um objeto JavaScript, adiciona a nova propriedade ao objeto e retorna o objeto atualizado.

Exercício 107: Filtrando Propriedades ao Converter para JSON

Descrição:
Use o segundo parâmetro da função JSON.stringify() para filtrar propriedades.

Solução:

```
function filtrarEConverter(obj) {
    return JSON.stringify(obj, ['nome', 'idade']);
}

const objetoTeste = { nome: "João", senha: "12345", idade: 25 };
console.log(filtrarEConverter(objetoTeste));
// "{'nome':'João','idade':25}"
```

Explicação:

A função filtrarEConverter usa o método JSON.stringify() para converter um objeto em uma string JSON, mas o segundo parâmetro da função é usado para especificar quais propriedades incluir na string resultante.

Exercício 108: Formatando a Saída JSON

Descrição:

Utilize o terceiro parâmetro da função JSON.stringify() para adicionar espaços de identação.

Solução:

```
function  formatarJson (obj) {
    return  JSON .stringify(obj, null , 2 );
}

const  objetoTeste = { nome: "João" , idade: 25  };
console .log(formatarJson(objetoTeste));
// "{
//   'nome': 'João',
//   'idade': 25
// }"
```

Explicação:

A função formatarJson converte o objeto em uma string JSON formatada com 2 espaços de identação, usando o terceiro parâmetro de JSON.stringify().

Exercício 109: Trabalhando com Arrays em JSON

Descrição:

Converta a string com JSON.parse(), adicione um novo objeto ao array e retorne o array atualizado.

Solução:

```
function  adicionarAoArray (jsonStr, novoObjeto) {
    const  arr = JSON .parse(jsonStr);
   arr.push(novoObjeto);
    return  arr;
}

const  jsonStringTeste = '[{"nome":"João","idade":25},
{"nome":"Maria","idade":30}]' ;
console .log(adicionarAoArray(jsonStringTeste, { nome: "Pedro" , idade: 22
 }));
// [
//   {
//     nome: "João",
//     idade: 25,
//   },
//   {
//     nome: "Maria",
//     idade: 30,
//   },
//   {
//     nome: "Pedro",
//     idade: 22,
//   },
// ];
```

Explicação:

A função adicionarAoArray converte a string JSON em um array JavaScript, adiciona o novo objeto ao array e retorna o array atualizado.

Exercício 110: Manipulando Valores ao Converter para JSON

Descrição:

Use o segundo parâmetro da função JSON.stringify() para alterar valores.

Solução:

```
function  alterarEConverter(obj) {
    const  alterarValor = (key, value) => {
        if  (key === "idade") return  value + 1;
        return  value;
   }
    return  JSON.stringify(obj, alterarValor);
}
const  objetoTeste = { nome: "João", idade: 25  };
console.log(alterarEConverter(objetoTeste));
// "{'nome':'João','idade':26}"
```

Explicação:

A função alterarEConverter utiliza a função alterarValor como o segundo parâmetro de JSON.stringify(). Esta função modifica o valor da propriedade "idade", adicionando 1 à idade original.

Capítulo 5: ES6 e Além

A evolução do JavaScript nunca para. Desde a sua criação em 1995, esta linguagem tem crescido e se transformado constantemente, adaptando-se às necessidades de desenvolvedores e às demandas das aplicações modernas. Uma das atualizações mais significativas foi a introdução do ECMAScript 2015, comumente conhecido como ES6. Esta versão trouxe consigo uma variedade de novos recursos, sintaxes e metodologias que não só tornaram o JavaScript mais poderoso, mas também mais eficiente e legível.

ES6 foi um marco para a linguagem, estabelecendo o padrão para futuras atualizações. Desde então, a cada ano, novas versões do ECMAScript têm sido lançadas, trazendo melhorias incrementais e novas funcionalidades.

Neste capítulo, vamos mergulhar nos recursos mais influentes do ES6 e de versões posteriores do ECMAScript. Abordaremos:

Let e Const: Novas formas de declarar variáveis que oferecem escopo de bloco e imutabilidade.
Arrow Functions: Uma nova e mais concisa sintaxe para funções.
Classes: Uma abstração mais clara sobre protótipos, facilitando a criação de objetos e herança.
Promises e Async/Await: Tornando o código assíncrono mais gerenciável e legível.
Desestruturação: Uma maneira elegante de extrair valores de objetos e arrays.
Parâmetros Rest e Spread: Trabalhando com quantidades indefinidas de argumentos em funções.
E muitos outros recursos...

Cada uma destas características trouxe novas possibilidades e otimizações. Se você é novo no ES6 ou deseja aprofundar seu conhecimento sobre as atualizações recentes do JavaScript, este capítulo é para você. Vamos explorar juntos o moderno mundo do JavaScript!

Template Literals

No mundo dinâmico e em constante evolução do JavaScript, a introdução dos Template Literals no ES6 representou um salto significativo na maneira como lidamos com strings. Antes deles, concatenar strings e variáveis ou expressões era, por vezes, confuso e verboso. Com os Template Literals, esse processo tornou-se mais intuitivo, legível e conciso.

Os Template Literals não são apenas uma forma mais elegante de construir strings. Eles também nos permitem criar strings multilinhas sem recorrer a truques ou hacks e embutir expressões diretamente dentro de uma string. Além disso, ao utilizar tagged templates, podemos até mesmo personalizar a construção da string de acordo com as nossas necessidades.

Nesta seção, vamos mergulhar no poder e na flexibilidade dos Template Literals, explorando seus recursos e aprendendo como eles podem tornar o nosso código mais limpo e eficiente.

Exercício 111: Crie uma função que aceite nome, sobrenome e idade e retorne uma apresentação usando template literals.

Descrição: A função deve retornar uma frase do tipo "Olá, meu nome é [nome] [sobrenome] e tenho [idade] anos." usando template literals.

Solução:

```
function apresentacao(nome, sobrenome, idade) {
    return `Olá, meu nome é ${nome} ${sobrenome} e tenho ${idade} anos.`;
}

console.log(apresentacao("João", "Silva", 30));  // Saída: "Olá, meu nome é João Silva e tenho 30 anos."
```

Explicação: Utilizamos template literals para embutir diretamente as variáveis nome, sobrenome e idade dentro da string resultante.

Exercício 112: Escreva uma função que aceite dois valores e retorne um texto que faça uma comparação entre eles usando template literals.

Descrição: A função deve retornar "O valor [valorA] é [maior/menor/igual] ao valor [valorB].".

Solução:

```
function comparaValores(valorA, valorB) {
    let comparacao = valorA === valorB ? "igual" : (valorA > valorB ? "maior" : "menor");
    return `O valor ${valorA} é ${comparacao} ao valor ${valorB}.`;
}

console.log(comparaValores(5, 3));  // Saída: "O valor 5 é maior ao valor 3."
console.log(comparaValores(2, 2));  // Saída: "O valor 2 é igual ao valor 2."
```

Explicação: Com base em uma expressão condicional ternária, determinamos a relação entre valorA e valorB e, em seguida, usamos template literals para construir a string final.

Exercício 113: Crie uma função que aceite uma lista de itens e retorne uma string enumerada usando template literals.

Descrição: A partir de um array como ['maçã', 'banana', 'uva'], a função deve retornar: "1. maçã, 2. banana, 3. uva".

Solução:

```
function listaEnumerada(itens) {
    return itens.map((item, index) => `${index + 1}. ${item}`).join(', ');
```

```
}
console.log(listaEnumerada([ 'maçã' , 'banana' , 'uva' ]));  // Saída: "1. maçã, 2. banana, 3. uva"
```

Explicação: Utilizamos o método map para criar um novo array com strings formatadas usando template literals. Depois, join é usado para transformar o array em uma única string.

Exercício 114: Crie uma função que aceite um objeto com informações de um livro (título, autor e ano) e retorne uma descrição do livro usando template literals.

Descrição: A partir de um objeto como { titulo: 'O Principe', autor: 'Maquiavel', ano: 1532 }, a função deve retornar: "O livro 'O Principe' foi escrito por Maquiavel em 1532."

Solução:

```
function descricaoLivro(livro) {
    return `O livro '${livro.titulo}' foi escrito por ${livro.autor} em ${livro.ano}.`;
}
console.log(descricaoLivro({ titulo: 'O Principe', autor: 'Maquiavel', ano: 1532 }));  // Saída: "O livro 'O Principe' foi escrito por Maquiavel em 1532."
```

Explicação: Através do acesso às propriedades do objeto livro, conseguimos construir uma descrição completa usando template literals.

Arrow Functions

Com a evolução da linguagem JavaScript e a introdução do ES6 (ECMAScript 2015), várias melhorias foram apresentadas para tornar o código mais limpo, conciso e legível. Uma dessas inovações é a Arrow Function.

As arrow functions oferecem uma sintaxe mais curta quando comparadas às funções tradicionais. Elas são especialmente úteis para funções simples e de linha única. Além da sintaxe reduzida, as arrow functions têm um comportamento diferente para o this, o que as torna ideais para certas situações, especialmente em funções de callback e métodos de array como map, filter e reduce.

Nesta seção, exploraremos a elegância e os benefícios de usar arrow functions, além de

algumas de suas peculiaridades e considerações importantes a serem lembradas ao trabalhar com elas. Prepare-se para simplificar e modernizar seu código JavaScript!

Exercício 115: Converta a seguinte função em uma arrow function e retorne o resultado.

```
function  helloWorld (name) {
    return  "Hello, "  + name + "!" ;
}
```

Descrição:

Dado uma função tradicional que aceita um nome como argumento e retorna uma saudação, sua tarefa é convertê-la em uma arrow function.

Solução:

```
const  helloWorld = name => `Hello, ${name}!` ;

// Testes
console .log(helloWorld( "Alice" ));   // Esperado: "Hello, Alice!"
console .log(helloWorld( "Bob" ));     // Esperado: "Hello, Bob!"
```

Explicação:

A arrow function permite uma sintaxe mais limpa e concisa. Usamos o template literal (introduzido no ES6) para interpolar a string, o que torna o código ainda mais limpo e fácil de ler.

Exercício 116: Escreva uma arrow function que aceite dois números como argumentos e retorne sua multiplicação.

Descrição:

Crie uma função simples que multiplique dois números.

Solução:

```
const  multiply = (a, b) => a * b;

// Testes
console .log(multiply( 5 , 4 ));   // Esperado: 20
console .log(multiply( 3 , 7 ));   // Esperado: 21
```

Explicação:

A arrow function toma dois parâmetros e simplesmente retorna o produto deles. Como é

uma expressão simples, o retorno é implícito.

Exercício 117: Escreva uma arrow function que não receba argumentos e retorne a data atual.

Descrição:

Crie uma função que retorne o dia de hoje.

Solução:

```
const today = () => new Date().toDateString();

// Testes
console.log(today());   // Retorna a data atual no formato: "Wed Jul 26 2023"
```

Explicação:

A função arrow não recebe nenhum argumento (indicado por ()), e retorna a data atual formatada como uma string legível.

Exercício 118: Crie uma arrow function que tome um array de números como argumento e retorne a soma de todos eles.

Descrição:

Dado um array de números, retorne a soma total.

Solução:

```
const sumArray = numbers => numbers.reduce((acc, curr) => acc + curr, 0);

// Testes
console.log(sumArray([1, 2, 3, 4]));   // Esperado: 10
console.log(sumArray([5, 5, 5]));     // Esperado: 15
```

Explicação:

A função utiliza o método reduce dos arrays para acumular a soma. A arrow function é usada dentro do reduce para simplificar a lógica de soma.

Exercício 119: Escreva uma arrow function que retorne uma mensagem

personalizada, dadas as propriedades "nome" e "idade" de um objeto.

Descrição:

Dado um objeto com propriedades "nome" e "idade", retorne uma mensagem formatada.

Solução:

```
const personalizedMessage = ({ name, age }) => `Hello, my name is ${name} and I am ${age} years old.` ;

// Testes
console.log(personalizedMessage({ name: "Alice", age: 30 }));   // Esperado: "Hello, my name is Alice and I am 30 years old."
console.log(personalizedMessage({ name: "Bob", age: 25 }));     // Esperado: "Hello, my name is Bob and I am 25 years old."
```

Explicação:

Esta arrow function utiliza a desestruturação de objetos para obter as propriedades "name" e "age" diretamente como argumentos. Depois, usamos os template literals para formatar e retornar a mensagem.

Exercício 120: Escreva uma arrow function que aceite um número como argumento e retorne uma função que incrementa esse número.

Descrição:

O objetivo deste exercício é familiarizar-se com a capacidade das arrow functions de retornar outras funções, criando um fechamento (closure).

Solução:

```
const incrementer = base => () => ++base;

// Testes
const incrementBy5 = incrementer(5);
console.log(incrementBy5());   // Esperado: 6
console.log(incrementBy5());   // Esperado: 7
```

Explicação:

A função externa incrementer recebe um número e retorna uma função interna sem parâmetros. Esta função interna, quando chamada, incrementará o número fornecido à função externa devido ao conceito de fechamento (closure).

Exercício 121: Escreva uma arrow function que aceite uma string e retorne outra arrow function que aceite outra string. A função resultante deve concatenar as duas strings.

Descrição:

Neste exercício, você trabalhará com arrow functions retornando outras arrow functions.

Solução:

```
const  concatFirstString = firstString => secondString => firstString +
secondString;

// Testes
const  helloString = concatFirstString( "Hello " );
console .log(helloString( "World!" ));   // Esperado: "Hello World!"
console .log(helloString( "Alice!" ));   // Esperado: "Hello Alice!"
```

Explicação:

A função concatFirstString pega uma string e retorna outra função que, quando chamada com uma segunda string, concatena a primeira e a segunda strings.

Parâmetros Default, Rest e Spread

Com a evolução da linguagem JavaScript, várias características foram adicionadas para facilitar o desenvolvimento e tornar o código mais legível e conciso. Entre essas características, destacam-se os Parâmetros Default, Rest e o operador Spread. Essas adições proporcionam maior flexibilidade na definição de funções, permitindo que os desenvolvedores manipulem argumentos de maneira mais eficiente e expressiva.

Parâmetros Default: Essa funcionalidade permite definir valores padrão para os parâmetros de uma função. Isso significa que, se um valor não for fornecido para um determinado parâmetro, o valor padrão será usado em seu lugar. É uma maneira elegante de lidar com argumentos opcionais, sem ter que recorrer a verificações adicionais no corpo da função.

Parâmetro Rest: É uma maneira de representar um número indefinido de argumentos como um array. Isso é particularmente útil quando você não sabe quantos argumentos serão passados para sua função, ou quando você quer coletar os argumentos adicionais em um array.

Operador Spread: O oposto do parâmetro rest, o operador spread permite que um array seja expandido em lugares onde zero ou mais argumentos são esperados, ou um

objeto em lugares onde zero ou mais pares de chave-valor são esperados.

Nesta seção, exploraremos essas funcionalidades em detalhes através de exemplos práticos. Esses recursos, apesar de simples, podem mudar profundamente a maneira como você aborda problemas e constrói soluções em JavaScript.

Exercício 122: Crie uma função que receba até três parâmetros, sendo os dois primeiros obrigatórios e o terceiro opcional. Se o terceiro parâmetro não for fornecido, atribua o valor "Desconhecido" a ele utilizando parâmetros default.

```
function  informacoes (nome, idade, cidade = "Desconhecido" ) {
    return  `Nome: ${nome}, Idade: ${idade}, Cidade: ${cidade}` ;
}

// Testes
console .log(informacoes( "Carlos" , 25 ));           // Nome: Carlos, Idade:
25, Cidade: Desconhecido
console .log(informacoes( "Ana" , 30 , "São Paulo" )); // Nome: Ana, Idade: 30,
Cidade: São Paulo
```

Explicação: O código define uma função chamada informacoes que aceita três parâmetros. O terceiro parâmetro, cidade, tem um valor default de "Desconhecido". Se a função é chamada sem fornecer um valor para cidade, o valor default é usado.

Exercício 123: Desenvolva uma função que utilize o parâmetro rest para receber um número indefinido de argumentos e retorne a soma de todos eles.

```
function  soma (...nums) {
    return  nums.reduce((acc, curr) => acc + curr, 0 );
}

// Testes
console .log(soma( 1 , 2 , 3 , 4 ));   // 10
console .log(soma( 5 , 5 , 10 ));     // 20
```

Explicação: A função soma utiliza o parâmetro rest para coletar todos os argumentos em um array chamado nums. Então, utiliza o método reduce para somar todos os números contidos nesse array.

Exercício 124: Implemente uma função que aceite um objeto e use o operador spread para criar uma cópia profunda desse objeto.

```
function copiaProfunda(obj) {
    return {...obj};
}

// Testes
const original = { nome: "Lucas", idade: 20 };
const copia = copiaProfunda(original);
console.log(copia);             // { nome: 'Lucas', idade: 20 }
original.nome = "Rafael";
console.log(copia.nome);        // Lucas
```

Explicação: A função copiaProfunda utiliza o operador spread para criar uma nova cópia do objeto passado como argumento. A modificação feita no objeto original após a cópia não afeta o objeto copiado.

Exercício 125: Escreva uma função que combine dois arrays em um novo array. Use o operador spread para realizar essa tarefa.

```
function combinarArrays(arr1, arr2) {
    return [...arr1, ...arr2];
}

// Testes
console.log(combinarArrays([1, 2, 3], [4, 5, 6]));  // [1, 2, 3, 4, 5,
6]
```

Explicação: A função combinarArrays utiliza o operador spread duas vezes para espalhar os elementos dos dois arrays fornecidos como argumentos em um novo array. O resultado é a combinação dos dois arrays.

Exercício 126: Crie uma função que aceite múltiplos argumentos utilizando o parâmetro rest, e retorne um novo array contendo apenas os argumentos que são números.

```
function filtrarNumeros(...args) {
    return args.filter(arg => typeof arg === "number");
}

// Testes
```

```
console.log(filtrarNumeros(1, "a", 3, "b", 5));  // [1, 3, 5]
```

Explicação: A função filtrarNumeros utiliza o parâmetro rest para coletar todos os argumentos em um array. Em seguida, utiliza o método filter para retornar um novo array contendo apenas os argumentos que são números.

Exercício 127: Implemente uma função que aceite dois objetos. Use o operador spread para mesclar os dois objetos em um novo objeto.

Descrição: O exercício ilustra como o operador spread pode ser usado para mesclar dois objetos. O resultado é um novo objeto que combina propriedades de ambos os objetos fornecidos.

```
function mesclarObjetos(obj1, obj2) {
    return {...obj1, ...obj2};
}

// Testes
const objeto1 = { nome: "Lucas" };
const objeto2 = { idade: 30 };
console.log(mesclarObjetos(objeto1, objeto2));  // { nome: "Lucas", idade: 30 }
```

Explicação: A função mesclarObjetos utiliza o operador spread duas vezes para espalhar as propriedades dos dois objetos fornecidos como argumentos em um novo objeto. O resultado é a combinação das propriedades dos dois objetos.

Exercício 128: Crie uma função que utilize parâmetros default para criar um objeto "carro" com propriedades marca, modelo e ano, onde apenas modelo é obrigatório.

Descrição: O foco deste exercício é o uso de parâmetros default para estabelecer valores padrão para funções. A função ilustra como podemos definir valores padrão para propriedades quando elas não são fornecidas na chamada da função.

```
function criarCarro(modelo, marca = "Desconhecido", ano = new Date().getFullYear()) {
    return { marca, modelo, ano };
}

// Testes
console.log(criarCarro("Civic"));                    // { marca: "Desconhecido", modelo: "Civic", ano: 2023 }
```

```
console.log(criarCarro("Civic", "Honda", 2020));     // { marca: "Honda",
modelo: "Civic", ano: 2020 }
```

Explicação: A função criarCarro aceita três parâmetros e usa parâmetros default para os parâmetros marca e ano. Se não forem fornecidos valores para marca e ano, eles receberão "Desconhecido" e o ano atual, respectivamente.

Exercício 129: Implemente uma função que receba dois arrays e retorne um novo array contendo os elementos do primeiro array seguido dos elementos do segundo array. A função deve usar o parâmetro rest e o operador spread.

Descrição: Neste exercício, a ênfase é colocada na combinação do parâmetro rest para coletar múltiplos argumentos e do operador spread para combinar arrays. Ele demonstra como podemos coletar vários arrays e combiná-los em um único array.

```
function unirArrays(...arrays) {
    return [].concat(...arrays);
}

// Testes
console.log(unirArrays([1, 2], [3, 4], [5, 6]));  // [1, 2, 3, 4, 5, 6]
```

Explicação: A função unirArrays utiliza o parâmetro rest para coletar todos os arrays fornecidos em um único array. Em seguida, utiliza o operador spread juntamente com concat para criar um novo array com todos os elementos dos arrays fornecidos.

Exercício 130: Crie uma função que use o operador spread para transformar uma string em um array de caracteres.

Descrição: O operador spread pode ser usado não apenas em arrays e objetos, mas também em strings, para transformá-las em arrays de caracteres individuais.

```
function stringParaArray(str) {
    return [...str];
}

// Testes
console.log(stringParaArray("Hello"));  // ["H", "e", "l", "l", "o"]
```

Explicação: A função stringParaArray usa o operador spread para espalhar os

caracteres da string fornecida em um novo array. O resultado é um array contendo todos os caracteres da string original.

Exercício 131: Implemente uma função que aceite dois arrays. O primeiro array é uma lista de objetos com uma chave e valor, e o segundo é uma lista de chaves. A função deve retornar um novo array de objetos contendo apenas as chaves especificadas no segundo array.

Descrição: Este exercício desafia os alunos a filtrar objetos com base em um conjunto especificado de chaves. Através disso, eles aprenderão sobre a manipulação de objetos e a importância de iterar corretamente sobre eles para alcançar o resultado desejado.

```
function  filtrarPorChaves (arrayObjetos, chaves) {
    return  arrayObjetos.map(obj => {
        let  novoObj = {};
       chaves.forEach(chave => {
            if (obj[chave] !== undefined ) novoObj[chave] = obj[chave];
       });
        return  novoObj;
   });
}
// Testes
const  data = [{ nome: "Lucas" , idade: 30 , cidade: "São Paulo"  }, { nome:
"Ana" , idade: 25  }];
console .log(filtrarPorChaves(data, [ "nome" , "cidade" ]));   // [{ nome:
"Lucas", cidade: "São Paulo" }, { nome: "Ana" }]
```

Explicação: A função filtrarPorChaves utiliza o método map para percorrer o array de objetos e, para cada objeto, percorre o array de chaves usando forEach. Apenas as chaves presentes no objeto e especificadas no array de chaves são incluídas no novo objeto.

Desestruturação (Destructuring)

A evolução constante das linguagens de programação muitas vezes revela recursos que não apenas tornam a codificação mais eficiente, mas também mais legível e concisa. A desestruturação, introduzida no ECMAScript 6, é um desses recursos notáveis para JavaScript. Em sua essência, permite-nos "desmontar" estruturas de dados, como objetos e arrays, em variáveis individuais, de uma maneira que é ao mesmo tempo intuitiva e poderosa.

Antes desse recurso, se quiséssemos acessar valores dentro de um objeto ou array, teríamos que acessá-los um por um, o que pode ser verboso, especialmente quando lidamos com objetos aninhados ou arrays. Com a desestruturação, podemos extrair múltiplas propriedades ou valores de uma vez, tornando nosso código mais limpo e mais declarativo.

A desestruturação não é apenas uma ferramenta de conveniência; ela estabelece um padrão que promove a clareza e a eficiência, permitindo que os desenvolvedores trabalhem com dados de forma mais direta e menos propensa a erros.

Exercício 132: Extração Simples de Propriedades de um Objeto

Descrição: Dado um objeto com várias propriedades, extraia apenas o nome e a idade desse objeto e retorne-os em uma string formatada.

```
function extrairDadosPessoa(pessoa) {
    const { nome, idade } = pessoa;
    return `Nome: ${nome}, Idade: ${idade}`;
}

// Testes
const pessoa = { nome: "Lucas", idade: 30, cidade: "São Paulo" };
console.log(extrairDadosPessoa(pessoa)); // Nome: Lucas, Idade: 30
```

Explicação: Utilizamos a desestruturação para extrair o nome e a idade do objeto pessoa. Em seguida, retornamos esses valores formatados em uma string.

Exercício 133: Desestruturação de Arrays com Valores Padrão

Descrição: Dado um array de números, extraia os três primeiros valores. Caso algum valor não esteja presente, ele deve ser substituído por null.

```
function extrairValores(array) {
    const [primeiro = null, segundo = null, terceiro = null] = array;
    return [primeiro, segundo, terceiro];
}

// Testes
console.log(extrairValores([1, 2])); // [1, 2, null]
console.log(extrairValores([1, 2, 3, 4])); // [1, 2, 3]
```

Explicação: Usamos desestruturação para extrair os três primeiros valores do array. Se algum valor não estiver presente, ele é atribuído como null.

Exercício 134: Desestruturação de Objetos Aninhados

Descrição: Dado um objeto que tem uma propriedade chamada endereco, que por sua vez é outro objeto contendo a propriedade cidade, extraia apenas a cidade desse objeto aninhado.

```
function extrairCidade(usuario) {
    const { endereco: { cidade } } = usuario;
    return cidade;
}

// Testes
const usuario = { nome: "Lucas", endereco: { rua: "Rua A", cidade: "São
Paulo", estado: "SP" } };
console.log(extrairCidade(usuario)); // São Paulo
```

Explicação: Aqui, desestruturamos a propriedade cidade do objeto aninhado endereco no objeto usuario.

Exercício 135: Desestruturação em Parâmetros de Função

Descrição: Dado um objeto com propriedades nome e idade, escreva uma função que desestruture essas propriedades diretamente em seus parâmetros e retorne uma string formatada.

```
function exibirInformacoes({ nome, idade }) {
    return `Nome: ${nome}, Idade: ${idade}`;
}

// Testes
const pessoa2 = { nome: "Ana", idade: 25 };
console.log(exibirInformacoes(pessoa2)); // Nome: Ana, Idade: 25
```

Explicação: Desestruturamos o objeto diretamente nos parâmetros da função, permitindo que trabalhemos diretamente com suas propriedades.

Exercício 136: Combinando Desestruturação com Rest

Descrição: Escreva uma função que aceite vários números como argumentos. Extraia os três primeiros números e coloque os restantes em um array separado.

```
function separarNumeros(primeiro, segundo, terceiro, ...resto) {
    return [primeiro, segundo, terceiro, resto];
```

```
}

// Testes
console.log(separarNumeros(1, 2, 3, 4, 5, 6)); // [1, 2, 3, [4, 5, 6]]
```

Explicação: Utilizamos a desestruturação para extrair os três primeiros números e o operador rest para coletar os números restantes em um array separado.

Exercício 137: Troca de Valores Usando Desestruturação

Descrição: Sem usar uma variável temporária, troque os valores de duas variáveis.

```
let a = 5;
let b = 10;
[a, b] = [b, a];

// Testes
console.log(a); // 10
console.log(b); // 5
```

Explicação: Usando desestruturação, podemos facilmente trocar os valores de a e b sem a necessidade de uma variável temporária.

Exercício 138: Desestruturação de Arrays Aninhados

Descrição: Dado um array de arrays, extraia o segundo valor do primeiro array interno e o primeiro valor do segundo array interno.

```
function extrairValoresAninhados(array) {
    const [[, valor1], [valor2]] = array;
    return [valor1, valor2];
}

// Testes
console.log(extrairValoresAninhados([[1, 2, 3], [4, 5]])); // [2, 4]
```

Explicação: Usamos desestruturação em arrays aninhados para extrair o segundo valor do primeiro array interno e o primeiro valor do segundo array interno.

Exercício 139: Desestruturação com Renomeação de Variáveis

Descrição: Dado um objeto com propriedades a e b, desestruture essas propriedades

renomeando-as para alpha e beta, respectivamente.

```
function  renomearVariaveis(obj) {
    const  { a: alpha, b: beta } = obj;
    return  { alpha, beta };
}

// Testes
const  objeto = { a: "valorA" , b: "valorB"  };
console.log(renomearVariaveis(objeto)); // { alpha: "valorA", beta: "valorB"
}
```

Explicação: Durante a desestruturação, podemos renomear as variáveis para nomes diferentes dos nomes das propriedades do objeto. Aqui, renomeamos a para alpha e b para beta.

Classes e Herança em JavaScript

O paradigma de programação orientada a objetos (POO) revolucionou a maneira como desenvolvemos software, permitindo modelar entidades do mundo real como objetos dentro do código. No centro desse paradigma estão as classes, que podem ser entendidas como moldes a partir dos quais os objetos são criados. Estes moldes não apenas definem propriedades (frequentemente chamadas de atributos) e comportamentos (métodos) dos objetos, mas também permitem a organização e reutilização eficiente do código.

JavaScript, sendo uma linguagem dinâmica e versátil, inicialmente adotou um sistema prototípico para implementar conceitos de POO. No entanto, com a introdução do ES6 (ECMAScript 2015), a linguagem agora também suporta uma sintaxe mais clássica para classes, proporcionando uma forma mais familiar de definir e criar objetos, especialmente para desenvolvedores vindos de outras linguagens orientadas a objetos, como Java ou C#.

Outro conceito fundamental da POO é a herança, que permite a criação de uma nova classe com base em uma classe existente. Através da herança, a nova classe pode herdar propriedades e métodos da classe base, permitindo a reutilização e extensão do código de maneira ordenada. Em JavaScript, a herança é implementada através da palavra-chave extends.

Nesta seção, mergulharemos profundamente nas classes e na herança em JavaScript. Exploraremos como definir classes, criar instâncias de objetos, estender classes e muito mais.

Exercício 140 - Classe Tempo

Descrição: Crie uma classe Tempo com propriedades para horas, minutos e segundos. Adicione métodos para somar e subtrair outro objeto Tempo.

```
class  Tempo  {
    constructor (horas = 0, minutos = 0, segundos = 0) {
        this .horas = horas;
        this .minutos = minutos;
        this .segundos = segundos;
   }

   somar(tempo) {
        let  totalSegundos = this .segundos + tempo.segundos;
        let  totalMinutos = this .minutos + tempo.minutos;
        let  totalHoras = this .horas + tempo.horas;

        if  (totalSegundos >= 60 ) {
           totalMinutos += Math .floor(totalSegundos / 60 );
           totalSegundos %= 60 ;
       }
        if  (totalMinutos >= 60 ) {
           totalHoras += Math .floor(totalMinutos / 60 );
           totalMinutos %= 60 ;
       }

        return  new  Tempo(totalHoras, totalMinutos, totalSegundos);
   }

   subtrair(tempo) {
        let  totalSegundos = this .segundos - tempo.segundos;
        let  totalMinutos = this .minutos - tempo.minutos;
        let  totalHoras = this .horas - tempo.horas;

        if  (totalSegundos < 0 ) {
           totalMinutos -= Math .ceil( Math .abs(totalSegundos) / 60 );
           totalSegundos = 60  - ( Math .abs(totalSegundos) % 60 );
       }
        if  (totalMinutos < 0 ) {
           totalHoras -= Math .ceil( Math .abs(totalMinutos) / 60 );
           totalMinutos = 60  - ( Math .abs(totalMinutos) % 60 );
       }

        return  new  Tempo(totalHoras, totalMinutos, totalSegundos);
   }
}

// Testes
const  tempo1 = new  Tempo( 1 , 20 , 30 );
const  tempo2 = new  Tempo( 0 , 50 , 40 );
let  resultado = tempo1.somar(tempo2);
console .log(
```

```
`${resultado.horas}h:${resultado.minutos}m:${resultado.segundos}s` ); //
2h:11m:10s

resultado = tempo1.subtrair(tempo2);
console .log(
`${resultado.horas}h:${resultado.minutos}m:${resultado.segundos}s` ); //
0h:29m:50s
```

Explicação: Criamos a classe Tempo com três propriedades: horas, minutos e segundos. Os métodos somar e subtrair recebem outro objeto Tempo como parâmetro. Convertemos tudo para segundos, realizamos a operação matemática desejada e depois retornamos à representação padrão de horas, minutos e segundos.

Exercício 141 - Interface usando Classes

Descrição: Simulando interfaces, crie duas classes, Voador e Nadador, cada uma com um método, voar e nadar, respectivamente. Em seguida, crie uma classe Pato que "implemente" ambas as "interfaces", fazendo a classe Pato ter os métodos das duas classes anteriores.

```
class  Voador  {
   voar() {
        return  "Estou voando!" ;
   }
}

class  Nadador  {
   nadar() {
        return  "Estou nadando!" ;
   }
}

class  Pato  extends  Voador  {
    constructor () {
        super ();
        // "Implementando" o método nadar do Nadador
        const  nadador = new  Nadador();
        this .nadar = nadador.nadar.bind( this );
   }
}

// Testes
const  pato = new  Pato();
console .log(pato.voar()); // Estou voando!
console .log(pato.nadar()); // Estou nadando!
```

Explicação: A classe Pato estende a classe Voador, herdando seu método voar. Para "implementar" o método nadar da classe Nadador, criamos uma instância de Nadador dentro do construtor de Pato e associamos o método nadar a this. O uso de bind garante

que o contexto de this seja o objeto Pato ao invocar o método.

Exercício 142 - Herança e Polimorfismo

Descrição: Crie uma classe Animal com um método falar que retorna "O animal faz um som". Em seguida, crie duas subclasses: Cão e Gato. Ambas devem sobrescrever o método falar, retornando "O cão late" e "O gato mia", respectivamente.

```
class Animal {
   falar() {
        return "O animal faz um som";
   }
}

class Cão extends Animal {
   falar() {
        return "O cão late";
   }
}

class Gato extends Animal {
   falar() {
        return "O gato mia";
   }
}

// Testes
const cão = new Cão();
console.log(cão.falar()); // O cão late

const gato = new Gato();
console.log(gato.falar()); // O gato mia
```

Explicação: A classe Animal define um método falar. As classes Cão e Gato estendem Animal e sobrescrevem o método falar para proporcionar sua própria implementação. Isso é um exemplo de polimorfismo.

Exercício 143 - Classe Abstrata

Descrição: Crie uma classe abstrata chamada Forma com um método abstrato area. Crie duas subclasses: Círculo e Retângulo, que implementam o método area.

```
class Forma {
   area() {
        throw new Error("Método 'area' deve ser implementado por
subclasses!");
```

```
    }
}

class Círculo extends Forma {
    constructor(raio) {
        super();
        this.raio = raio;
    }

    area() {
        return Math.PI * this.raio * this.raio;
    }
}

class Retângulo extends Forma {
    constructor(largura, altura) {
        super();
        this.largura = largura;
        this.altura = altura;
    }

    area() {
        return this.largura * this.altura;
    }
}

// Testes
const círculo = new Círculo(5);
console.log(círculo.area()); // 78.53981633974483

const retângulo = new Retângulo(4, 6);
console.log(retângulo.area()); // 24
```

Explicação: A classe Forma simula ser uma classe abstrata ao lançar um erro no método area. As subclasses Círculo e Retângulo implementam esse método para calcular suas respectivas áreas.

Exercício 144 - Encapsulamento

Descrição: Crie uma classe Pessoa que encapsule a idade da pessoa, garantindo que a idade não possa ser negativa.

```
class Pessoa {
    constructor(nome, idade) {
        this.nome = nome;
        this._idade = idade;
    }

    get idade() {
        return this._idade;
    }
```

```
    set idade(valor) {
        if (valor < 0) {
            console.log("Idade não pode ser negativa!");
            return;
        }
        this._idade = valor;
    }
}

// Testes
const pessoa = new Pessoa("João", 30);
console.log(pessoa.idade); // 30

pessoa.idade = -5; // Idade não pode ser negativa!
console.log(pessoa.idade); // 30
```

Explicação: A classe Pessoa usa o método get e set para criar um acessador de propriedade chamado idade. Isso permite validar os valores definidos e garantir que a idade nunca seja negativa.

Exercício 145 - Extensão de Métodos

Descrição: Crie uma classe Vendedor que estenda uma classe Pessoa. O Vendedor deve ter um método vender e uma propriedade totalVendas, que é incrementada a cada venda.

```
class Pessoa {
    constructor(nome) {
        this.nome = nome;
    }

    falar() {
        return `${this.nome} está falando.`;
    }
}

class Vendedor extends Pessoa {
    constructor(nome) {
        super(nome);
        this.totalVendas = 0;
    }

    vender() {
        this.totalVendas++;
        return `${this.nome} fez uma venda.`;
    }
}

// Testes
const vendedor = new Vendedor("Carlos");
```

```
console.log(vendedor.vender()); // Carlos fez uma venda.
console.log(vendedor.totalVendas); // 1
```

Explicação: A classe Vendedor estende a classe Pessoa. Ele herda o método falar e adiciona seu próprio método vender e a propriedade totalVendas.

Exercício 146 - Classe Estática

Descrição: Crie uma classe Calculadora que tenha um método estático somar, que aceite dois números e retorne sua soma.

```
class Calculadora {
    static somar(a, b) {
        return a + b;
    }
}

// Testes
console.log(Calculadora.somar(5, 3)); // 8

    Explicação: A classe Calculadora tem um método estático chamado somar.
Métodos estáticos pertencem à classe e não à instância da classe. Eles são
chamados no próprio nome da classe, como demonstrado no teste.
```

Exercício 147 - Método de Fábrica

Descrição: Crie uma classe Carro que possui um método de fábrica criarSedan que retorna uma nova instância de um carro com a propriedade tipo definida como 'Sedan'.

```
class Carro {
    constructor(tipo) {
        this.tipo = tipo;
    }

    static criarSedan() {
        return new Carro('Sedan');
    }
}

// Testes
const sedan = Carro.criarSedan();
console.log(sedan.tipo); // Sedan
```

Explicação: A classe Carro tem um método estático chamado criarSedan, que atua como um método de fábrica. Ele cria e retorna uma nova instância da classe Carro com o tipo definido como 'Sedan'.

Exercício 148 - Classes e Exceções

Descrição: Crie uma classe Conta com um método sacar. Se o valor do saque for maior que o saldo disponível, lance uma exceção.

```
class Conta {
    constructor(saldo) {
        this.saldo = saldo;
    }

    sacar(valor) {
        if (valor > this.saldo) {
            throw new Error('Saldo insuficiente.');
        }
        this.saldo -= valor;
        return this.saldo;
    }
}

// Testes
const conta = new Conta(100);
console.log(conta.sacar(50)); // 50

try {
    conta.sacar(100);
} catch (e) {
    console.log(e.message); // Saldo insuficiente.
}
```

Explicação: A classe Conta tem um método sacar que verifica se o valor de saque é maior que o saldo disponível. Se for, ele lança uma exceção, caso contrário, ele deduz o valor do saldo e retorna o novo saldo.

Exercício 149 - Classe com Métodos Privados

Descrição: Crie uma classe Segredo que tenha um método privado chamado _mensagemSecreta e um método público para acessar essa mensagem.

```
class Segredo {
    #mensagemSecreta = "Esta é uma mensagem secreta!";

    revelarSegredo() {
        return this.#mensagemSecreta;
    }
}

// Testes
const segredo = new Segredo();
```

```
console.log(segredo.revelarSegredo()); // Esta é uma mensagem secreta!
```

Explicação: A classe Segredo contém um campo privado #mensagemSecreta. Esse campo só pode ser acessado dentro da própria classe. O método público revelarSegredo retorna essa mensagem.

Exercício 150 - Herança e Construtores

Descrição: Crie uma classe Veículo com um construtor que aceite marca e modelo. Em seguida, crie uma classe Carro que herde de Veículo e adicione um construtor que aceite marca, modelo e numPortas.

```
class Veículo {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }
}

class Carro extends Veículo {
    constructor(marca, modelo, numPortas) {
        super(marca, modelo);
        this.numPortas = numPortas;
    }
}

// Testes
const carro = new Carro("Toyota", "Corolla", 4);
console.log(carro); // { marca: 'Toyota', modelo: 'Corolla', numPortas: 4 }
```

Explicação: A classe Carro estende a classe Veículo. No construtor do Carro, chamamos o construtor da classe pai usando super(marca, modelo) e depois inicializamos a propriedade numPortas.

Promises e Async/Await

Em um mundo onde as operações assíncronas são uma realidade inegável, especialmente em ambientes como desenvolvimento web, ter ferramentas eficientes para gerenciar a asincronicidade torna-se essencial. JavaScript, como linguagem de programação predominantemente usada na web, tem evoluído ao longo dos anos para fornecer abstrações robustas e elegantes para lidar com operações assíncronas: as Promises e o Async/Await.

Promises oferecem uma maneira de representar operações que ainda não foram

concluídas, mas que são esperadas no futuro. Elas proporcionam uma abordagem mais flexível e compreensível para lidar com o resultado ou erro de uma operação assíncrona do que as tradicionais callbacks. Uma Promise em JavaScript é um objeto que representa o eventual cumprimento ou falha de uma operação assíncrona.

Por outro lado, Async/Await é um açúcar sintático que foi introduzido para tornar o código assíncrono em JavaScript ainda mais legível, quase fazendo-o parecer sincronizado. Ele nos permite trabalhar com Promises de uma maneira mais direta, sem ter que depender de métodos como .then() e .catch().

Exercício 151: Implemente uma função que retorne uma Promise que resolve após um tempo específico em milissegundos.

Descrição: Este exercício testa a habilidade de criar uma Promise que utiliza o setTimeout para resolver após um determinado período.

Solução com testes:

```
function delay(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

// Teste
delay(1000).then(() => console.log('Resolvido após 1 segundo!'));
```

Explicação: A função delay retorna uma Promise que se resolve após um tempo determinado por ms graças ao setTimeout. A função setTimeout é chamada com a função resolve da Promise e o tempo em milissegundos, garantindo que a Promise seja resolvida após esse período.

Exercício 152: Crie uma função assíncrona que simule a busca de dados de um banco de dados e lance um erro se a entrada for uma string específica.

Descrição: O objetivo é simular uma operação assíncrona que pode falhar com base em determinadas entradas.

Solução com testes:

```
async function fetchData(input) {
    if (input === "error") {
        throw new Error("Entrada inválida!");
   }
    return `Dados para ${input}`;
}
```

```
// Teste
fetchData( "data" )
   .then(data => console .log(data))
   .catch(err => console .error(err.message));

fetchData( "error" )
   .then(data => console .log(data))
   .catch(err => console .error(err.message));
```

Explicação: A função fetchData verifica se a entrada é a string "error". Se for, ela lança um erro; caso contrário, ela retorna os dados. Como é uma função assíncrona, ela retornará implicitamente uma Promise.

Exercício 153: Use o Promise.all para aguardar várias Promises e retornar seus resultados em um array.

Descrição: Testa o entendimento sobre como aguardar múltiplas Promises simultaneamente.

Solução com testes:

```
function  promise1 () {
    return  new  Promise (resolve => setTimeout(() => resolve( "Resultado 1"
), 1000 ));
}

function  promise2 () {
    return  new  Promise (resolve => setTimeout(() => resolve( "Resultado 2"
), 1500 ));
}

// Teste
Promise .all([promise1(), promise2()])
   .then(results => console .log(results))
   .catch(err => console .error(err.message));
```

Explicação: As funções promise1 e promise2 retornam Promises que se resolvem após diferentes intervalos de tempo. Usando Promise.all, esperamos que ambas as Promises se resolvam e retornem seus resultados em um array.

Exercício 154: Implemente uma função assíncrona que use try/catch para lidar com erros ao esperar uma Promise.

Descrição: Esse exercício visa aprimorar a habilidade de lidar com erros em funções

assíncronas.

Solução com testes:

```
async function handleErrors(promise) {
    try {
        const result = await promise;
        console.log(result);
    } catch (error) {
        console.error(`Erro: ${error.message}`);
    }
}

// Testes
const successfulPromise = Promise.resolve("Sucesso!");
const failingPromise = Promise.reject(new Error("Falha!"));

handleErrors(successfulPromise);   // Deve imprimir "Sucesso!"
handleErrors(failingPromise);     // Deve imprimir "Erro: Falha!"
```

Explicação: A função handleErrors aceita uma Promise como argumento. Dentro de um bloco try/catch, ela espera a resolução da Promise usando await. Se a Promise for resolvida com sucesso, o resultado é impresso. Se a Promise for rejeitada, o erro é capturado e impresso.

Exercício 155: Implemente uma função async que simule a operação de busca de dados em uma API. Se o status da resposta for 200, a função deve retornar os dados, caso contrário, deve lançar um erro.

Descrição: Este exercício visa aprimorar a habilidade de fazer uma chamada de API usando async/await e de tratar possíveis erros.

Solução com testes:

```
async function fetchDataFromAPI(url) {
    let response = await fetch(url);
    if (response.status !== 200) {
        throw new Error("Falha na obtenção dos dados");
    }
    return await response.json();
}

// Teste (este é apenas um exemplo, e o URL pode precisar ser modificado)
fetchDataFromAPI("https://api.example.com/data")
    .then(data => console.log(data))
    .catch(err => console.error(err.message));
```

Explicação: Utilizamos o fetch para fazer a chamada de API e depois verificamos o

status da resposta. Se o status não for 200, lançamos um erro. Caso contrário, retornamos os dados em formato JSON.

Exercício 156: Crie uma função que aguarde várias Promises e retorne a primeira que for resolvida.

Descrição: Esse exercício testa o entendimento de como trabalhar com várias Promises e retornar o valor da primeira que for resolvida.

Solução com testes:

```
function firstResolvedPromise(promises) {
    return Promise.race(promises);
}

// Teste
const p1 = new Promise(resolve => setTimeout(() => resolve("Primeira"),
500));
const p2 = new Promise(resolve => setTimeout(() => resolve("Segunda"),
1000));

firstResolvedPromise([p1, p2]).then(result => console.log(result)); // Deve
imprimir "Primeira"
```

Explicação: Promise.race é usado para retornar a primeira Promise que for resolvida (ou rejeitada) entre as Promises fornecidas.

Exercício 157: Crie uma função async que simule a recuperação de dois recursos de API em sequência, onde o segundo requer dados do primeiro.

Descrição: Este exercício testa a capacidade de fazer chamadas sequenciais de API usando async/await.

Solução com testes:

```
async function fetchSequentialData(firstURL, secondURLGenerator) {
    let firstData = await fetchDataFromAPI(firstURL);
    let secondURL = secondURLGenerator(firstData);
    return await fetchDataFromAPI(secondURL);
}

// Teste
// Suponha que a primeira API retorne um ID que é necessário para a segunda
chamada de API
fetchSequentialData("https://api.example.com/first", data =>
`https://api.example.com/second/${data.id}`)
```

```
    .then(result => console.log(result))
    .catch(error => console.error(error.message));
```

Explicação: Fazemos uma chamada de API para firstURL e usamos os dados retornados para gerar a URL da segunda chamada de API. As chamadas são feitas em sequência.

Exercício 158: Utilize o async/await dentro de um método de classe para simular o carregamento de dados para essa classe.

Descrição: Testa a integração do async/await dentro de classes.

Solução com testes:

```
class DataLoader {
    constructor(apiURL) {
        this.apiURL = apiURL;
        this.data = null;
    }

    async loadData() {
        this.data = await fetchDataFromAPI(this.apiURL);
    }
}

// Teste
const loader = new DataLoader("https://api.example.com/data");
loader.loadData()
    .then(() => console.log(loader.data))
    .catch(err => console.error(err.message));
```

Explicação: A classe DataLoader tem um método loadData que usa async/await para carregar dados de uma API e armazená-los na propriedade data da classe.

Exercício 159: Crie uma função que retorna uma Promise que rejeita após um tempo específico em milissegundos.

Descrição: Este exercício visa testar a capacidade de criar Promises que podem ser rejeitadas após um certo tempo.

Solução com testes:

```
function rejectAfterDelay(ms) {
    return new Promise((_, reject) => setTimeout(() => reject(new Error(
"Rejeitado após timeout!")), ms));
```

```
}

// Teste
rejectAfterDelay( 1000 ).catch(err => console .error(err.message)); // Deve
imprimir "Rejeitado após timeout!"
```

Explicação: Semelhante ao exercício de resolução após um atraso, mas neste caso, rejeitamos a Promise após o período especificado.

Exercício 160: Utilize async/await em um loop para processar uma série de chamadas de API em sequência.

Descrição: Esse exercício testa a habilidade de usar async/await dentro de loops.

Solução com testes:

```
async  function  processInSequence (urls) {
    let  results = [];
    for  ( let  url of  urls) {
        let  data = await  fetchDataFromAPI(url);
       results.push(data);
   }
    return  results;
}

// Teste
const  urls = [ "https://api.example.com/data1" ,
"https://api.example.com/data2" ];
processInSequence(urls)
   .then(data => console .log(data))
   .catch(err => console .error(err.message));
```

Explicação: Usamos um loop for...of para processar uma série de URLs em sequência. Para cada URL, esperamos que os dados sejam carregados antes de continuar para a próxima.

Iteradores e Geradores

No mundo dinâmico da programação JavaScript, muitas vezes nos encontramos em situações onde precisamos trabalhar com coleções de dados ou sequências de valores que não são necessariamente armazenados em estruturas de dados tradicionais como arrays ou objetos. Aqui é onde entram os iteradores e geradores, duas ferramentas poderosas e flexíveis introduzidas no ECMAScript 6 (ES6) que nos permitem manipular e criar sequências de dados de maneiras novas e inovadoras.

Iteradores são objetos que permitem iterar sobre elementos de uma coleção, seja ela um array, uma string ou outra estrutura de dados. Eles fornecem uma interface consistente para a iteração, permitindo que você acesse itens um de cada vez sem necessariamente conhecer a estrutura subjacente da coleção.

Por outro lado, **geradores** são funções especiais que podem pausar sua execução e posteriormente retomar de onde pararam. Isso permite que eles produzam sequências de valores sob demanda, tornando-os particularmente úteis para representar sequências infinitas ou para simular lazy evaluation (avaliação preguiçosa) — um mecanismo onde os valores são computados apenas quando são realmente necessários.

Exercício 161: Crie um iterador que gera uma sequência Fibonacci.

Descrição: Um iterador Fibonacci deve retornar os números na sequência Fibonacci sempre que o método next for chamado.

Solução:

```
function * fibonacci () {
    let [prev, curr] = [ 0 , 1 ];
    while (true) {
        yield curr;
       [prev, curr] = [curr, prev + curr];
   }
}

// Testes
const fib = fibonacci();
console.log(fib.next().value); // 1
console.log(fib.next().value); // 1
console.log(fib.next().value); // 2
console.log(fib.next().value); // 3
```

Explicação:
A função geradora produz valores da sequência Fibonacci indefinidamente. Começamos com os dois primeiros números da sequência (0 e 1). A cada chamada do next, nós retornamos o valor atual e atualizamos nossas variáveis para o próximo número da sequência.

Exercício 162: Utilize um gerador para criar um iterável que gere uma sequência aritmética, dado um valor inicial, uma razão e o número de termos.

Descrição: O gerador deve aceitar três parâmetros e produzir uma sequência aritmética baseada nos inputs fornecidos.

Solução:

```
function * arithmeticSequence (start, reason, n) {
    for ( let  i = 0 ; i < n; i++) {
        yield  start;
       start += reason;
   }
}
// Testes
const  sequence = arithmeticSequence( 1 , 2 , 5 );
console .log([...sequence]); // [1, 3, 5, 7, 9]
```

Explicação:
A função geradora começa com o valor inicial e adiciona a razão a cada iteração até que a sequência tenha o tamanho desejado.

Exercício 163: Implemente um gerador que atue como uma paginação para um array, retornando os itens em lotes.

Descrição: Dado um array e um tamanho de lote, o gerador deve retornar sub-arrays (lotes) do tamanho especificado até que todos os itens sejam retornados.

Solução:

```
function * paginate (array, batchSize) {
    for ( let  i = 0 ; i < array.length; i += batchSize) {
        yield  array.slice(i, i + batchSize);
   }
}
// Testes
const  items = [ 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 ];
const  pages = paginate(items, 4 );
console .log(pages.next().value); // [1, 2, 3, 4]
console .log(pages.next().value); // [5, 6, 7, 8]
console .log(pages.next().value); // [9, 10]
```

Explicação:
Usamos o método slice para pegar lotes do array baseados no índice atual e no tamanho do lote.

Exercício 164: Crie um gerador que retorne números primos.

Descrição: O gerador deve produzir números primos indefinidamente sempre que o método next for chamado.

Solução:

```
function * primeGenerator () {
    function  isPrime (num) {
        for ( let  i = 2 ; i * i <= num; i++)
            if (num % i === 0 ) return  false ;
        return  num > 1 ;
   }

    let  num = 2 ;
    while  ( true ) {
        if  (isPrime(num)) yield  num;
       num++;
   }
}

// Testes
const  primes = primeGenerator();
console .log(primes.next().value); // 2
console .log(primes.next().value); // 3
console .log(primes.next().value); // 5
console .log(primes.next().value); // 7
```

Explicação:
A função isPrime é usada para verificar se um número é primo. O gerador começa em 2 (o primeiro número primo) e verifica cada número em sequência para determinar se é primo, retornando os que são.

Exercício 165: Crie um iterador que retorne os números pares e ímpares alternadamente de um array, começando pelos pares.

Descrição: Dado um array, o iterador deve retornar os números alternando entre pares e ímpares, começando pelos números pares.

Solução:

```
function * alternateEvenOdd (array) {
    let  evens = array.filter(x => x % 2  === 0 );
    let  odds = array.filter(x => x % 2  !== 0 );
    let  index = 0 ;

    while  (index < evens.length || index < odds.length) {
        if  (index < evens.length) yield  evens[index];
        if  (index < odds.length) yield  odds[index];
```

```
        index++;
    }
}

// Testes
const numbers = [1, 2, 3, 4, 5, 6, 7, 8];
const alternated = alternateEvenOdd(numbers);
console.log([...alternated]); // [2, 1, 4, 3, 6, 5, 8, 7]
```

Explicação:
Primeiro, separamos os números em pares e ímpares usando o método filter. Em seguida, alternamos entre os dois arrays para produzir os números desejados.

Exercício 166: Implemente um gerador que produza uma sequência geométrica.

Descrição: Dado um valor inicial a, uma razão r, e um número de termos n, produza uma sequência geométrica.

Solução:

```
function* geometricSequence(a, r, n) {
    for (let i = 0; i < n; i++) {
        yield a;
        a *= r;
    }
}

// Testes
const sequence = geometricSequence(1, 2, 5);
console.log([...sequence]); // [1, 2, 4, 8, 16]
```

Explicação:
A função geradora começa com o valor inicial a e multiplica pelo valor da razão r a cada iteração, até que a sequência atinja o tamanho desejado.

Exercício 167: Crie um gerador que reproduza o comportamento da função range do Python.

Descrição: A função range em Python aceita até 3 argumentos: start, stop, e step. Crie um gerador que imite essa funcionalidade.

Solução:

```
function* range(start, stop = null, step = 1) {
```

```
    if (stop === null) {
        stop = start;
        start = 0;
    }

    for (let i = start; i < stop; i += step) {
        yield i;
    }
}

// Testes
console.log([...range(5)]);       // [0, 1, 2, 3, 4]
console.log([...range(2, 5)]);    // [2, 3, 4]
console.log([...range(0, 10, 2)]); // [0, 2, 4, 6, 8]
```

Explicação:
Esta função geradora mimetiza a funcionalidade da função range do Python. Se apenas um valor for passado, ele será tratado como o valor de parada, com o início padrão em 0. Caso contrário, ela seguirá os argumentos de início, parada e passo fornecidos.

Exercício 168: Use um gerador para criar um iterador que reproduza o comportamento da função zip do Python.

Descrição: A função zip em Python aceita múltiplos iteráveis e retorna um iterador que produz tuplas contendo os elementos dos iteráveis de entrada, emparelhados.

Solução:

```
function* zip(...iterables) {
    const iterators = iterables.map(iterable => iterable[Symbol.iterator]());
    while (true) {
        const items = iterators.map(iterator => iterator.next());
        if (items.some(item => item.done)) break;
        yield items.map(item => item.value);
    }
}

// Testes
const a = [1, 2, 3];
const b = ['a', 'b', 'c'];
console.log([...zip(a, b)]); // [[1, 'a'], [2, 'b'], [3, 'c']]
```

Explicação:
Criamos um array de iteradores para cada iterável de entrada. Em cada iteração, recuperamos o próximo valor de cada iterador e verificamos se algum dos iteradores foi concluído. Se um iterador estiver concluído, encerramos o loop. Caso contrário, retornamos uma tupla de valores.

Exercício 169: Crie um gerador que gere uma série Fibonacci até um número máximo n.

Descrição: O gerador deve produzir a série Fibonacci até que o valor ultrapasse um limite n fornecido.

Solução:

```
function * fibonacci (n) {
    let [prev, curr] = [ 0 , 1 ];
    while (curr <= n) {
        yield curr;
       [prev, curr] = [curr, prev + curr];
   }
}
// Testes
console .log([...fibonacci( 100 )]); // [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Explicação:
Utilizamos a desestruturação de arrays para trocar facilmente os valores prev e curr a cada iteração, produzindo assim a série Fibonacci. O loop continua até que o valor ultrapasse o limite n fornecido.

Exercício 170: Implemente um gerador que simule a funcionalidade de enumerate do Python.

Descrição: A função enumerate do Python retorna um iterador que produz tuplas contendo índices e valores de um iterável.

Solução:

```
function * enumerate (iterable) {
    let index = 0 ;
    for ( const value of iterable) {
        yield [index, value];
       index++;
   }
}
// Testes
const letters = [ 'a' , 'b' , 'c' ];
console .log([...enumerate(letters)]); // [[0, 'a'], [1, 'b'], [2, 'c']]
```

Explicação:
O gerador itera sobre cada valor do iterável de entrada, retornando uma tupla contendo

o índice atual e o valor.

Exercício 171: Crie um gerador que retorne uma sequência aritmética.

Descrição: Dado um valor inicial a, uma diferença d, e um número de termos n, produza uma sequência aritmética.

Solução:

```
function * arithmeticSequence (a, d, n) {
    for  ( let  i = 0 ; i < n; i++) {
        yield  a;
       a += d;
   }
}

// Testes
const  sequence = arithmeticSequence( 1 , 2 , 5 );
console .log([...sequence]); // [1, 3, 5, 7, 9]
```

Explicação:
A função geradora começa com o valor inicial a e adiciona a diferença d a cada iteração, até que a sequência atinja o tamanho desejado.

Métodos de Array ES6+

Desde a introdução do ECMAScript 2015 (também conhecido como ES6) e em versões subsequentes do JavaScript, uma série de novos métodos para manipulação de arrays foram adicionados. Estes métodos não apenas proporcionam uma maneira mais concisa e expressiva de trabalhar com arrays, mas também tornam algumas tarefas mais intuitivas e menos propensas a erros.

Se antes tínhamos de criar loops para encontrar um elemento dentro de um array ou verificar se todos os elementos de um array satisfazem uma determinada condição, agora temos métodos que nos permitem fazer essas operações de maneira declarativa.

Nesta seção, exploraremos alguns desses métodos, como find, findIndex, includes, some, every, from e outros. Cada um deles tem sua própria utilidade e otimiza diferentes aspectos da manipulação de arrays. Conhecer e entender esses métodos é essencial para qualquer desenvolvedor que deseje escrever códigos mais limpos, eficientes e modernos em JavaScript. Vamos embarcar nesta jornada e descobrir o poder e a elegância dos métodos de array disponíveis em ES6 e versões mais recentes!

Exercício 172: Filtro Ímpar

Descrição: Dado um array de números, utilize o método filter para retornar apenas os números ímpares.

Solução com testes:

```
function filterOdd(numbers) {
    return numbers.filter(n => n % 2 !== 0);
}

console.log(filterOdd([1, 2, 3, 4, 5])); // [1, 3, 5]
console.log(filterOdd([10, 12, 15]));   // [15]
```

Explicação: A função filterOdd faz uso do método filter para retornar todos os números ímpares no array. O método filter cria um novo array com todos os elementos que passam no teste da função fornecida.

Exercício 173: Primeiro Maior que Dez

Descrição: Usando o método find, procure o primeiro número no array que seja maior que 10.

Solução com testes:

```
function findGreaterThanTen(numbers) {
    return numbers.find(n => n > 10);
}

console.log(findGreaterThanTen([1, 2, 12, 8, 15])); // 12
```

Explicação: A função findGreaterThanTen usa o método find para retornar o primeiro número que é maior que 10. O método find retorna o primeiro valor no array que passa no teste fornecido.

Exercício 174: Índice do Elemento "JavaScript"

Descrição: Utilize o método findIndex para encontrar o índice da primeira ocorrência da string "JavaScript" em um array de strings.

Solução com testes:

```
function findJavaScriptIndex(strings) {
    return strings.findIndex(s => s === "JavaScript");
```

```
    }

    console.log(findJavaScriptIndex(["Java", "Python", "JavaScript", "C++"
])); // 2
```

Explicação: A função utiliza o método findIndex para retornar o índice da primeira ocorrência da string "JavaScript". O método findIndex retorna o índice do primeiro elemento no array que passa no teste fornecido.

Exercício 175: Todos Positivos

Descrição: Verifique usando o método every se todos os elementos de um array de números são positivos.

Solução com testes:

```
function areAllPositive(numbers) {
    return numbers.every(n => n > 0);
}

console.log(areAllPositive([1, 2, 3, 4, 5])); // true
console.log(areAllPositive([-1, 2, 3, 4, 5])); // false
```

Explicação: A função areAllPositive utiliza o método every para verificar se todos os números são positivos. O método every testa se todos os elementos no array passam pelo teste fornecido.

Exercício 176: Transformação com map

Descrição: Dado um array de números, utilize o método map para dobrar todos os valores do array.

Solução com testes:

```
    function doubleValues(numbers) {
        return numbers.map(n => n * 2);
    }

    console.log(doubleValues([1, 2, 3, 4]));   // [2, 4, 6, 8]
```

Explicação: A função doubleValues usa o método map para dobrar cada número no array. O método map cria um novo array com os resultados de chamar uma função para cada elemento do array.

Exercício 177: Verificar Existência com includes

Descrição: Dado um array e um elemento, utilize o método includes para verificar se o elemento existe no array.

Solução com testes:

```
function doesInclude(array, element) {
    return array.includes(element);
}

console.log(doesInclude(['a', 'b', 'c'], 'b'));  // true
console.log(doesInclude(['a', 'b', 'c'], 'z'));  // false
```

Explicação: A função doesInclude utiliza o método includes para verificar a existência de um elemento no array. includes retorna true se o array contiver o elemento e false caso contrário.

Exercício 178: some Menores que Dez

Descrição: Dado um array de números, verifique se pelo menos um dos números é menor que 10 utilizando o método some.

Solução com testes:

```
function hasSmallerThanTen(numbers) {
    return numbers.some(n => n < 10);
}

console.log(hasSmallerThanTen([15, 20, 25]));  // false
console.log(hasSmallerThanTen([5, 20, 25]));  // true
```

Explicação: A função hasSmallerThanTen usa o método some para verificar se pelo menos um número no array é menor que 10. O método some testa se ao menos um dos elementos no array atende ao teste fornecido.

Exercício 179: Array a Partir de Strings com from

Descrição: Converta uma string em um array de caracteres usando o método from.

Solução com testes:

```
    function  stringToArray (str) {
        return  Array .from(str);
    }

    console .log(stringToArray( 'hello' ));   // ['h', 'e', 'l', 'l', 'o']
```

Explicação: A função stringToArray usa o método from para converter uma string em um array de caracteres. O método from cria uma nova instância de Array a partir de um objeto iterável.

Exercício 180: Concatenando Arrays com concat

Descrição: Concatene dois arrays usando o método concat.

Solução com testes:

```
function  concatArrays (arr1, arr2) {
    return  arr1.concat(arr2);
}

console .log(concatArrays([ 1 , 2 ], [ 3 , 4 ]));   // [1, 2, 3, 4]
```

Explicação: A função concatArrays utiliza o método concat para concatenar dois arrays. O método concat retorna um novo array que é a junção dos arrays fornecidos.

Exercício 181: Utilizando find para buscar um objeto

Descrição: Dado um array de objetos com propriedade name e age, utilize o método find para buscar o primeiro objeto que possui a idade maior que 20.

Solução com testes:

```
    function  findOlderThan20 (people) {
        return  people.find(person => person.age > 20 );
    }

    const  samplePeople = [
       {name: 'Alice' , age: 18 },
       {name: 'Bob' , age: 21 },
       {name: 'Charlie' , age: 19 }
    ];

    console .log(findOlderThan20(samplePeople));   // {name: 'Bob', age: 21}
```

Explicação: A função findOlderThan20 usa o método find para buscar o primeiro objeto que atende ao critério fornecido (idade maior que 20). O método find retorna o

primeiro elemento que satisfaz a função de teste.

Exercício 182: Verificando com every

Descrição: Dado um array de números, verifique se todos os números são positivos utilizando o método every.

Solução com testes:

```
function areAllPositive(numbers) {
    return numbers.every(n => n > 0);
}

console.log(areAllPositive([1, 2, 3])); // true
console.log(areAllPositive([-1, 2, 3])); // false
```

Explicação: A função areAllPositive utiliza o método every para verificar se todos os números do array são positivos. O método every retorna true se todos os elementos no array atendem à função de teste e false caso contrário.

Exercício 183: Filtrando pares com filter

Descrição: Dado um array de números, retorne um novo array contendo apenas os números pares usando o método filter.

Solução com testes:

```
function filterEvens(numbers) {
    return numbers.filter(n => n % 2 === 0);
}

console.log(filterEvens([1, 2, 3, 4])); // [2, 4]
```

Explicação: A função filterEvens utiliza o método filter para criar um novo array contendo apenas os números pares do array original. O método filter cria um novo array com todos os elementos que atendem à função de teste.

xercício 184: Convertendo em String com join

Descrição: Dado um array de strings, converta-o em uma única string, onde cada palavra é separada por um espaço, usando o método join.

Solução com testes:

```
    function  joinWords (words) {
        return  words.join( ' ' );
   }

    console .log(joinWords([ 'Hello' , 'World' ]));   // 'Hello World'
```

Explicação: A função joinWords utiliza o método join para converter um array de strings em uma única string, separando as palavras com um espaço. O método join une todos os elementos de um array em uma string.

Exercício 185: Verificando a existência com findIndex

Descrição: Dado um array de números, encontre o índice do primeiro número que é divisível por 5 usando o método findIndex.

Solução com testes:

```
function  findDivisibleBy5 (numbers) {
    return  numbers.findIndex(n => n % 5  === 0 );
}

console .log(findDivisibleBy5([ 2 , 4 , 7 , 10 , 11 ]));   // 3
```

Explicação: A função findDivisibleBy5 usa o método findIndex para encontrar o índice do primeiro número que é divisível por 5 no array. O método findIndex retorna o índice do primeiro elemento que atende à função de teste, e -1 se nenhum elemento atender.

Capítulo 6: JavaScript no Navegador

Ao pensar em JavaScript, muitas pessoas automaticamente o associam ao desenvolvimento web, e por uma boa razão. A origem do JavaScript está firmemente enraizada em sua capacidade de adicionar interatividade e funcionalidades dinâmicas às páginas da web, transformando a experiência estática do HTML em algo muito mais envolvente.

Na seção "JavaScript no Navegador", mergulharemos na rica tapeçaria de capacidades que o JavaScript oferece quando executado no ambiente de um navegador web. Iremos explorar:

O Objeto Document: A espinha dorsal da manipulação da web com JavaScript, este objeto nos permite acessar e modificar elementos e atributos em uma página da web.

Eventos: Aprenderemos como JavaScript escuta e responde a uma variedade de ações

do usuário, como cliques, movimentos do mouse e pressionamentos de tecla.

Manipulação do DOM: O Document Object Model (DOM) é a representação estruturada de um documento web. Usando JavaScript, podemos alterar, adicionar ou remover elementos, transformando efetivamente a estrutura e a aparência de uma página.

Armazenamento Local e de Sessão: Descubra como armazenar informações diretamente no navegador do usuário, permitindo a criação de experiências personalizadas e a retenção de dados entre as sessões de navegação.

Obs: algumas soluções neste capítulo serão mais objetivas no código, em um sistema de maior escala e com diversos programadores a abordagem correta seria isolar os arquivos de HTML e JavaScript, além de separar os eventos do HTML.

Exercício 186: Dada uma página com uma lista não ordenada (<ul>) de itens, escreva uma função JavaScript que adicione um novo item ao final dessa lista.

Descrição:
O objetivo é selecionar uma lista não ordenada e inserir um novo item nessa lista. Para isso, você usará o document.querySelector para obter a referência da lista e então adicionar um novo item.

Solução com Testes:

```
<!-- HTML -->
< ul  id= "item-list" >
    < li > Item 1 </ li >
    < li > Item 2 </ li >
    < li > Item 3 </ li >
</ ul >
< button  onclick= "addItem()" > Add Item </ button >

< script >
    function  addItem () {
        const  ul = document .querySelector( '#item-list' );
        const  li = document .createElement( 'li' );
       li.textContent = `Item ${ul.children.length + 1 }` ;
       ul.appendChild(li);
   }
</ script >
```

Explicação:
Nesta solução, temos um botão que, quando clicado, chama a função addItem(). Dentro

desta função:

Selecionamos o elemento <ul> com o id "item-list" usando document.querySelector.
Criamos um novo elemento <li> com document.createElement.
Atribuímos um texto ao elemento <li>, baseado na quantidade atual de itens da lista.
Finalmente, adicionamos este novo <li> ao <ul> usando appendChild.

Exercício 187: Crie uma função que, ao clicar em um botão, altere a cor de fundo de um parágrafo para azul.

Descrição:
Ao clicar em um botão, você deve selecionar um elemento parágrafo e alterar sua cor de fundo para azul.

Solução com Testes:

```
<!-- HTML -->
< p  id= "text" > Este é um texto. </ p >
< button  onclick= "changeColor()" > Mudar Cor </ button >

< script >
    function  changeColor () {
        const  p = document .querySelector( '#text' );
       p.style.backgroundColor = 'blue' ;
   }
</ script >
```

Explicação:
Na solução, ao clicar no botão, a função changeColor() é chamada. Ela seleciona o parágrafo com id "text" e altera sua propriedade de estilo backgroundColor para 'blue'.

Exercício 188: Crie uma função que exiba em tempo real o que está sendo digitado em um campo de texto.

Descrição:
Cada vez que um usuário digitar algo em um campo de texto (<input type="text">), o conteúdo desse campo deve ser exibido em um elemento <p> abaixo dele.

Solução com Testes:

```
<!-- HTML -->
< input  type= "text"  id= "user-input" >
< p  id= "display-text" ></ p >

< script >
```

```
    const inputField = document.querySelector('#user-input');
    const displayText = document.querySelector('#display-text');

   inputField.addEventListener('input', function() {
       displayText.textContent = inputField.value;
   });
</script>
```

Explicação:
Usamos o método addEventListener para adicionar um ouvinte ao evento 'input' do campo de texto. Toda vez que o conteúdo do campo de texto muda, a função é chamada, atualizando o texto do parágrafo para refletir o valor atual do campo de texto.

Exercício 189: Ao clicar em um botão, adicione ou remova uma classe "ativo" de um elemento.

Descrição:
Você deve criar um botão que, quando clicado, alternará a classe "ativo" de um elemento . Se o elemento tiver a classe, ela deve ser removida; se não tiver, deve ser adicionada.

Solução com Testes:

```
<!-- HTML -->
<div id="toggle-element">Clique no botão para alternar minha classe!</div>
<button onclick="toggleClass()">Toggle Classe</button>

<script>
    function toggleClass() {
        const div = document.querySelector('#toggle-element');
       div.classList.toggle('ativo');
   }
</script>
```

Explicação:
Aqui, utilizamos o método classList.toggle do elemento. Esse método verificará se o elemento tem a classe especificada: se tiver, ele a removerá; se não tiver, ele a adicionará.

Exercício 190: Dada uma lista de itens, crie uma função que exiba apenas os itens que contêm uma determinada palavra.

Descrição:
Você terá uma lista não ordenada (<ul>) de itens. Abaixo dela, haverá um campo de texto e um botão. O usuário pode digitar uma palavra no campo e, ao clicar no botão, apenas os itens da lista que contêm essa palavra devem ser exibidos.

Solução com Testes:

```
<!-- HTML -->
< ul  id= "item-list" >
    < li > Maçã </ li >
    < li > Laranja </ li >
    < li > Mamão </ li >
    < li > Morango </ li >
</ ul >
< input  type= "text"  id= "filter-input" >
< button  onclick= "filterItems()" > Filtrar </ button >

< script >
    function  filterItems () {
        const  filter = document .querySelector( '#filter-input'
).value.toLowerCase();
        const  items = document .querySelectorAll( '#item-list li' );

       items.forEach(item => {
            if (item.textContent.toLowerCase().includes(filter)) {
               item.style.display = '' ;
           } else  {
               item.style.display = 'none' ;
           }
       });
   }
</ script >
```

Explicação:
Quando o botão é clicado, a função filterItems() é chamada. Essa função primeiro obtém o valor do campo de texto e o converte para minúsculas. Em seguida, seleciona todos os itens da lista e verifica se o texto de cada item contém o filtro. Se contiver, o item é exibido; caso contrário, ele é ocultado.

Exercício 191: Crie uma função que adicione uma classe "visível" a um elemento quando ele estiver visível na janela de visualização ao rolar a página.

Descrição:
Você terá um elemento que estará fora da janela de visualização (viewport) quando a página for carregada. Ao rolar a página e esse elemento entrar na janela de visualização, adicione a ele a classe "visível".

Solução com Testes:

```
<!-- HTML -->
< div  id= "scroll-element"  style= "margin-top: 120vh;" > Sou visível agora? </
div >

< script >
    window .addEventListener( 'scroll' , function () {
        const  div = document .querySelector( '#scroll-element' );
        const  rect = div.getBoundingClientRect();

        if  (rect.top <= window .innerHeight && rect.bottom >= 0 ) {
           div.classList.add( 'visivel' );
       } else  {
           div.classList.remove( 'visivel' );
       }
   });
</ script >
```

Explicação:
Adicionamos um ouvinte de evento para o evento de rolagem (scroll) na janela. Sempre que o usuário rolar a página, verificamos a posição do elemento usando getBoundingClientRect(). Se o elemento estiver dentro da janela de visualização, adicionamos a classe "visível". Caso contrário, removemos a classe.

Exercício 192: Crie um botão que, quando clicado, altere o tema da página entre claro e escuro.

Descrição:
Haverá um botão na página. Quando clicado, se o tema atual for claro, ele deve mudar para escuro e vice-versa.

Solução com Testes:

```
<!-- HTML -->
< button  onclick= "toggleTheme()" > Trocar Tema </ button >

< script >
    function  toggleTheme () {
        const  body = document .querySelector( 'body' );

        if  (body.classList.contains( 'dark-theme' )) {
           body.classList.remove( 'dark-theme' );
       } else  {
           body.classList.add( 'dark-theme' );
       }
   }
</ script >

< style >
```

```
    .dark-theme  {
        background-color : black;
        color : white;
   }
</ style >
```

Explicação:
A função toggleTheme() verifica se o <body> tem a classe dark-theme. Se tiver, ele a remove, tornando o tema claro; se não, ele adiciona a classe, tornando o tema escuro.

Exercício 193: Crie um botão que, quando clicado, exiba um modal na tela. O modal deve ter também um botão que o feche.

Descrição:
Haverá um botão na página principal. Quando clicado, um modal (uma pequena janela/overlay) deve aparecer no centro da tela. Esse modal terá um botão de fechar, que quando clicado, fechará o modal.

Solução com Testes:

```
<!-- HTML -->
< button  onclick= "showModal()" > Abrir Modal </ button >
< div  id= "modal"  style= "display: none; position: fixed; top: 50%; left: 50%; transform: translate(-50%, -50%); z-index: 10; background-color: white; padding: 20px;" >
   Isso é um modal!
    < button  onclick= "closeModal()" > Fechar </ button >
</ div >

< script >
    function  showModal () {
        const  modal = document .querySelector( '#modal' );
       modal.style.display = 'block' ;
   }

    function  closeModal () {
        const  modal = document .querySelector( '#modal' );
       modal.style.display = 'none' ;
   }
</ script >
```

Explicação:
O modal, por padrão, é ocultado com display: none. Ao clicar no botão para abrir o modal, a função showModal() é chamada e o modal é exibido alterando o estilo display para 'block'. O botão fechar no modal chama a função closeModal(), que oculta novamente o modal.

Exercício 194: Ao clicar em uma imagem na página, mostre essa imagem em tamanho ampliado em um modal.

Descrição:
Você terá algumas imagens miniaturas na página. Quando o usuário clicar em uma delas, a imagem deve ser exibida em tamanho maior em um modal. Também deve haver um botão para fechar o modal.

Solução com Testes:

```
<!-- HTML -->
< img  src= "thumbnail1.jpg"  alt= "Imagem 1"  onclick= "openImage('full1.jpg')"
/>
< img  src= "thumbnail2.jpg"  alt= "Imagem 2"  onclick= "openImage('full2.jpg')"
/>
< div  id= "image-modal"  style= "display: none; position: fixed; top: 0; left:
0; width: 100%; height: 100%; background-color: rgba(0,0,0,0.7); z-index:
10;" >
    < img  id= "modal-img"  src= ""  alt= "Imagem Ampliada"  style= "position:
absolute; top: 50%; left: 50%; transform: translate(-50%, -50%); max-width:
90%; max-height: 90%;" >
    < button  onclick= "closeImage()"  style= "position: absolute; top: 10px;
right: 10px;" > Fechar </ button >
</ div >

< script >
    function  openImage (src) {
        const  modal = document .querySelector( '#image-modal' );
        const  img = document .querySelector( '#modal-img' );
       img.src = src;
       modal.style.display = 'block' ;
   }

    function  closeImage () {
        const  modal = document .querySelector( '#image-modal' );
       modal.style.display = 'none' ;
   }
</ script >
```

Explicação:
As imagens miniaturas têm um onclick que chama a função openImage com a URL da imagem em tamanho completo. Quando clicado, a imagem em tamanho completo é carregada no modal, que é exibido. O botão de fechar oculta o modal.

Exercício 195: Crie um botão "Feedback" que, quando clicado, mostra um formulário modal para o usuário inserir feedback. Ao enviar, mostre um

alerta de agradecimento.

Descrição:
O usuário deve ser capaz de fornecer feedback por meio de um formulário que inclui um campo de texto. Depois de enviar, um alerta de agradecimento deve ser mostrado e o modal deve ser fechado.

Solução com Testes:

```
<!-- HTML -->
< button  onclick= "showForm()" > Feedback </ button >
< div  id= "feedback-modal"  style= "display: none; position: fixed; top: 50%;
left: 50%; transform: translate(-50%, -50%); z-index: 10; background-color:
white; padding: 20px;" >
    < textarea  id= "feedback-text"  placeholder= "Deixe seu feedback..." ></
textarea >< br />
    < button  onclick= "submitFeedback()" > Enviar </ button >
</ div >

< script >
    function  showForm () {
        const  modal = document .querySelector( '#feedback-modal' );
       modal.style.display = 'block' ;
   }

    function  submitFeedback () {
        const  feedback = document .querySelector( '#feedback-text' ).value;
        // Aqui, você pode enviar o feedback para o servidor ou processá-lo
como quiser.
       alert( "Obrigado pelo feedback!" );
       closeForm();
   }

    function  closeForm () {
        const  modal = document .querySelector( '#feedback-modal' );
       modal.style.display = 'none' ;
   }
</ script >
```

Explicação:
O botão de feedback abre o modal do formulário. Ao enviar o feedback, mostramos um alerta de agradecimento e fechamos o modal.

Exercício 196: Em um campo de texto, exiba um contador de caracteres restantes. Permita no máximo 150 caracteres.

Descrição:

Ao lado de um campo de texto, haverá uma contagem regressiva a partir de 150. À medida que o usuário digita, o contador deve diminuir. Se o usuário exceder 150 caracteres, mostre uma mensagem de erro.

Solução com Testes:

```
<!-- HTML -->
< textarea  id= "text-field"  oninput= "updateCounter()" ></ textarea >
< div > Caracteres restantes: < span  id= "counter" > 150 </ span ></ div >

< script >
    function  updateCounter () {
        const  textField = document .querySelector( '#text-field' );
        const  counter = document .querySelector( '#counter' );
        const  remaining = 150  - textField.value.length;

       counter.textContent = remaining;

        if  (remaining < 0 ) {
           counter.style.color = 'red' ;
           alert( "Você excedeu o limite de caracteres!" );
       } else  {
           counter.style.color = 'black' ;
       }
   }
</ script >
```

Explicação:
Cada vez que o usuário digita no campo de texto (oninput), chamamos a função updateCounter(). Esta função atualiza o contador e verifica se o limite de caracteres foi excedido.

Exercício 197: Crie um botão que altere o conteúdo de um título e de um parágrafo ao mesmo tempo.

Descrição:
O usuário deve ser capaz de clicar em um botão que alterne o conteúdo de um título e de um parágrafo simultaneamente.

Solução com Testes:

```
<!-- HTML -->
< h1  id= "titulo" > Olá, Mundo! </ h1 >
< p  id= "paragrafo" > Este é um parágrafo de exemplo. </ p >
< button  id= "botao" > Clique aqui </ button >

< script >
const  tituloElemento = document .getElementById( "titulo" );
const  paragrafoElemento = document .getElementById( "paragrafo" );
```

```
const  botaoElemento = document .getElementById( "botao" );

botaoElemento.addEventListener( "click" , function  () {
tituloElemento.textContent = "Título Modificado" ;
paragrafoElemento.textContent = "Parágrafo alterado após o clique!" ;
});
</ script >
```

Explicação:
Inicialmente, capturamos os nossos elementos da DOM, e em seguida, aguardamos o evento de click do botão e alteramos o conteúdo do título e do parágrafo..

Exercício 198: Implemente um dropdown personalizado usando HTML e JavaScript.

Descrição:
Em vez de usar o dropdown padrão <select>, crie um usando divs e listas. O dropdown deve exibir as opções quando clicado e permitir ao usuário selecionar uma opção.

Solução com Testes:

```
<!-- HTML -->
< div  class= "dropdown"  onclick= "toggleDropdown()" >
    < div  id= "selected-option" > Opção 1 </ div >
    < ul  id= "options-list"  style= "display: none;" >
        < li  onclick= "selectOption('Opção 1')" > Opção 1 </ li >
        < li  onclick= "selectOption('Opção 2')" > Opção 2 </ li >
        < li  onclick= "selectOption('Opção 3')" > Opção 3 </ li >
    </ ul >
</ div >

< script >
    function  toggleDropdown () {
        const  list = document .querySelector( '#options-list' );
       list.style.display = list.style.display === "none"  ? "block"  :
"none" ;
   }

    function  selectOption (option) {
        const  selected = document .querySelector( '#selected-option' );
       selected.textContent = option;
       toggleDropdown();   // fecha o dropdown após selecionar
   }
</ script >
```

Explicação:
Temos uma div que atua como nosso dropdown. Ao clicar nela, as opções são exibidas. Ao selecionar uma opção, atualizamos a div para mostrar a opção selecionada e fechamos o dropdown.

Exercício 199: Implemente uma funcionalidade de arrastar e soltar para reordenar uma lista.

Descrição:
Você terá uma lista de itens. O usuário deve ser capaz de reordenar essa lista arrastando e soltando itens.

Solução com Testes:

```
<!-- HTML -->
< ul  id= "draggable-list" >
    < li  draggable= "true"  ondragstart= "dragStart(event)"  ondrop=
"drop(event)"  ondragover= "allowDrop(event)" > Item 1 </ li >
    < li  draggable= "true"  ondragstart= "dragStart(event)"  ondrop=
"drop(event)"  ondragover= "allowDrop(event)" > Item 2 </ li >
    < li  draggable= "true"  ondragstart= "dragStart(event)"  ondrop=
"drop(event)"  ondragover= "allowDrop(event)" > Item 3 </ li >
</ ul >

< script >
    let  draggedItem = null ;

    function  dragStart (e) {
       draggedItem = e.target;
   }

    function  allowDrop (e) {
       e.preventDefault();
   }

    function  drop (e) {
       e.preventDefault();
        if  (e.target.tagName === "LI" ) {
           e.target.parentNode.insertBefore(draggedItem,
e.target.nextSibling);
       }
   }
</ script >
```

Explicação:
Usamos os eventos de drag and drop do HTML5. Quando um item é começado a ser arrastado, armazenamos ele em draggedItem. Quando ele é solto, nós o inserimos antes do item sobre o qual ele foi solto.

Exercício 200: Crie um guia de abas onde o conteúdo é alterado ao clicar nas abas.

Descrição:
Você terá várias abas na parte superior e conteúdo correspondente abaixo. Ao clicar em uma aba, apenas o conteúdo dessa aba deve ser exibido.

Solução com Testes:

```
<!-- HTML -->
< div  class= "tabs" >
    < button  onclick= "showTab('content1')" > Aba 1 </ button >
    < button  onclick= "showTab('content2')" > Aba 2 </ button >
</ div >
< div  id= "content1"  class= "tab-content" > Conteúdo da Aba 1 </ div >
< div  id= "content2"  class= "tab-content"  style= "display:none;" > Conteúdo da
Aba 2 </ div >

< script >
    function  showTab (id) {
        const  contents = document .querySelectorAll( '.tab-content' );
       contents.forEach(content => {
           content.style.display = 'none' ;
       });
        document .querySelector( '#'  + id).style.display = 'block' ;
   }
</ script >
```

Explicação:
Temos botões que representam abas e divs correspondentes para o conteúdo. Quando um botão é pressionado, ocultamos todo o conteúdo e exibimos apenas o conteúdo correspondente ao botão pressionado.

Capítulo 7: Regex

Expressões Regulares , frequentemente referidas como Regex ou RegExp, representam uma ferramenta poderosa e flexível para o processamento de texto. Originadas dos trabalhos teóricos em teoria dos autômatos e linguagens formais, elas encontraram aplicações práticas em quase todos os campos da computação, desde a validação simples de formatos de e-mail até a análise sintática de linguagens de programação.

A capacidade de regex reside na sua sintaxe concisa, que pode capturar padrões complexos em strings. Com um punhado de caracteres especiais, como ^, $, *, + e ?, e grupos definidos por [] e (), é possível definir padrões de busca que seriam extremamente verbosos (ou mesmo impraticáveis) com abordagens convencionais de string.

No entanto, com grande poder vem grande responsabilidade. Regex pode ser intrincado e, às vezes, confuso. A curva de aprendizado pode parecer íngreme para iniciantes, e é

comum ver iniciantes e até desenvolvedores experientes lutando com a sintaxe. No entanto, uma vez dominada, regex se torna uma ferramenta inestimável no arsenal de um desenvolvedor.

Nesta seção, vamos explorar o vasto mundo das expressões regulares, desde os conceitos básicos até as nuances mais avançadas.

Exercício 200: Dado uma string, verifique se ela é um e-mail válido.

Descrição: Usando regex, você deve identificar se a string fornecida corresponde ao formato típico de um endereço de e-mail.

Solução com testes:

```
function isValidEmail(email) {
    const regex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/;
    return regex.test(email);
}

// Testes
console.log(isValidEmail("example@email.com")); // true
console.log(isValidEmail("invalid.email@")); // false
```

Explicação: A regex verifica se o e-mail tem caracteres alfanuméricos antes do "@", seguido por mais caracteres alfanuméricos, um ponto e de dois a quatro caracteres alfabéticos após o ponto.

Exercício 20 1 : Extraia todas as URLs válidas de um texto fornecido.

Descrição: Dado um texto, extraia todas as strings que correspondem ao formato de uma URL.

Solução com testes:

```
function extractUrls(text) {
    const regex = /(https?:\/\/[^\s]+)/g;
    return text.match(regex) || [];
}

// Testes
console.log(extractUrls("Veja estes links https://www.example.com e
http://test.com")); // ["https://www.example.com", "http://test.com"]
```

Explicação: A regex busca por substrings que começam com "http" ou "https", seguido por "://", e continua até encontrar um espaço ou o final da string.

Exercício 202: Verifique se uma string corresponde a um formato de data válido (dd/mm/aaaa).

Descrição: Dado um texto, determine se ele é uma data válida no formato mencionado.

Solução com testes:

```
function isValidDate(date) {
  // Verificar formato da data
    const regex = /^(0[1-9]|[12][0-9]|3[01])\/(0[1-9]|1[0-2])\/\d{4}$/;
    if (!regex.test(date)) {
        return false;
   }

    // Verificar datas não existentes
    const [day, month, year] = date.split('/').map(Number);
    const testDate = new Date(year, month - 1, day);
    return testDate.getDate() === day && testDate.getMonth() + 1 === month;
}

// Testes
console.log(isValidDate("25/12/2020")); // true
console.log(isValidDate("31/02/2020")); // false
```

Explicação: A regex verifica se a data tem dois dígitos para o dia (01 a 31), dois dígitos para o mês (01 a 12) e quatro dígitos para o ano, a segunda parte verifica se a data é válida, por exemplo: 31/02 passa na regex mas não na última verificação.

Exercício 203: Substitua todas as ocorrências de tags HTML <tag> por [tag] em um texto fornecido.

Descrição: Converta tags HTML em uma notação com colchetes.

Solução com testes:

```
function replaceHtmlTags(text) {
    const regex = /<(\w+)>(.*?)<\/\1>/g;
    return text.replace(regex, '[$1]$2[/$1]');
}

// Testes
console.log(replaceHtmlTags("<div>Hello</div>")); // [div]Hello[/div]
```

Explicação: A regex captura o nome da tag HTML entre os símbolos "<" e ">". O método replace é usado para substituir a tag capturada pela notação de colchetes, além de acrescentar a "/" na tag de fechamento.

Exercício 204: Extraia todos os números de telefone válidos de um texto. Considere formatos como (xx) xxxxx-xxxx e xx-xxxxx-xxxx.

Descrição: Dado um texto, identifique e retorne uma lista de números de telefone válidos nele.

Solução com testes:

```
function extractPhoneNumbers(text) {
    const regex = /\(?\d{2}\)?[-.\s]?\d{4,5}[-.\s]?\d{4}/g;
    return text.match(regex) || [];
}

// Testes
console.log(extractPhoneNumbers("Contact at (12) 34567-8901 or 12-3456-7890."));
// ["(12) 34567-8901", "12-3456-7890"]
```

Explicação: A regex busca padrões que se assemelhem aos formatos de números de telefone mencionados, com ou sem parênteses e com diferentes separadores.

Exercício 205: Crie uma expressão regular para validar senhas. As senhas válidas devem ter pelo menos uma letra maiúscula, uma letra minúscula, um número e um caracter especial (como !, @, #, $, etc.). Elas devem ter pelo menos 8 caracteres e no máximo 20.

Descrição: Use uma regex para validar a força de uma senha conforme as condições fornecidas.

Solução com testes:

```
function isValidPassword(password) {
    const regex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,20}$/;
    return regex.test(password);
}

// Testes
console.log(isValidPassword("StrongPass1!")); // true
console.log(isValidPassword("weakpass")); // false
```

Explicação: A regex usa lookaheads positivos para verificar se a senha contém pelo menos uma letra minúscula, uma letra maiúscula, um dígito e um caractere especial. Ela então verifica o comprimento da senha.

Exercício 206: Remova todos os comentários HTML de uma string fornecida.

Descrição: Dada uma string contendo HTML, remova qualquer parte que seja um comentário HTML.

Solução com testes:

```
function removeHtmlComments(text) {
    const regex = /<!--[\s\S]*?-->/g;
    return text.replace(regex, '');
}

// Testes
console.log(removeHtmlComments("Hello <!-- This is a comment --> World"));
// "Hello  World"
```

Explicação: A regex captura comentários HTML, que começam com <!-- e terminam com -->. O método replace é usado para remover esses comentários.

Exercício 207: Verifique se um CEP é válido.

Descrição: Dado um determinando CEP, verifique se o mesmo é válido, podendo ter ou não um "-" entre os cinco primeiros e os três últimos dígitos.

Solução com testes:

```
function validateCEP(cep) {
    const cepRegex = /^\d{5}-?\d{3}$/;
    return cepRegex.test(cep);
}

// Testes:
console.log(validateCEP("12345-678")); // true
console.log(validateCEP("12345678")); // true
console.log(validateCEP("1234")); // false
```

Explicação: A regex verifica o CEP informado, e verifica a quantidade de números informados, podendo ter o "-" ou não, e retorna "true" ou "false".

Exercício 208: Extraia todas as tags e seus respectivos conteúdos de uma string HTML.

Descrição: Dada uma string HTML, extraia um objeto onde a chave é o nome da tag e o valor é o conteúdo dentro da tag.

Solução com testes:

```
function extractTagsAndContent(html) {
    const regex = /<(\w+)>([^<]+)<\/\1>/g;
    let match;
    const result = {};

    while ((match = regex.exec(html)) !== null) {
       result[match[1]] = match[2];
   }
    return result;
}

// Testes
console.log(extractTagsAndContent("<div>Hello</div><p>World</p>"));
// {
//  div: "Hello",
//  p: "World"
// }
```

Explicação: A regex busca por pares de tags de abertura e fechamento e extrai o nome da tag e o conteúdo. O resultado é acumulado em um objeto.

Exercício 209: Verifique se um texto possui palavras repetidas consecutivas.

Descrição: Dado um texto, determine se ele possui palavras que são repetidas consecutivamente, independentemente de serem maiúsculas ou minúsculas.

Solução com testes:

```
function hasConsecutiveRepeats(text) {
    const regex = /\b(\w+)\b\s+\b\1\b/i;
    return regex.test(text);
}

// Testes
console.log(hasConsecutiveRepeats("Hello hello world")); // true
console.log(hasConsecutiveRepeats("Hello world")); // false
```

Explicação: A regex verifica palavras consecutivas repetidas usando um grupo de captura e referenciando esse grupo novamente com \1. O modificador i torna a busca insensível a maiúsculas/minúsculas.

Exercício 210: Extraia o nome do domínio de uma URL.

Descrição: Dado uma URL, obtenha apenas o nome do domínio.

Solução com testes:

```
function  extractDomain (url) {
    const  regex = /^(?:https?:\/\/)?(?:www\.)?([^\/]+)/i ;
    const  match = url.match(regex);
    return  match ? match[ 1 ] : '' ;
}

// Testes
console .log(extractDomain( "https://www.example.com/page" ));
// "example.com"
```

Explicação: A regex captura o domínio da URL, ignorando o protocolo (como "http:" ou "https:") e a subparte "www." se estiver presente. O método match é usado para extrair o domínio.

Conclusão e próximos passos

Chegar ao final deste eBook é uma demonstração do seu comprometimento e dedicação à arte da programação. A linguagem JavaScript, com sua versatilidade e presença dominante no cenário da web, oferece uma vasta gama de oportunidades, e os 200 exercícios que você enfrentou aqui foram elaborados para ajudá-lo a aprofundar sua compreensão e habilidade técnica.

Cada seção deste livro foi cuidadosamente pensada para abordar diferentes aspectos do JavaScript, começando com os conceitos mais básicos até os mais avançados, como Promises e Async/Await, Regex e operações com o DOM. Ao resolver esses exercícios, você não apenas praticou sua habilidade de codificação, mas também desenvolveu um pensamento lógico e analítico, crucial para qualquer desenvolvedor.

Mas a jornada não para aqui. O mundo da tecnologia está em constante evolução. Novas bibliotecas, frameworks e padrões de design surgem frequentemente. Portanto, é essencial manter a chama da aprendizagem sempre acesa. Os fundamentos que você solidificou aqui irão servir como um alicerce sólido para explorar essas novas tendências e tecnologias.

Se você ainda não possui habilidades em bibliotecas e frameworks já existentes,

recomendo que invista nisso também. Há muitas ferramentas que são usadas em conjunto com o JavaScript para facilitar a rotina dos desenvolvedores. Você pode aprender algumas delas com os seguintes cursos: React do Zero a Maestria , Desafios de React , Node.js do Zero a Maestria , e TypeScript do Básico ao Avançado .

Além disso, encorajamos você a voltar e revisitar os exercícios sempre que sentir necessidade, talvez tentando abordagens diferentes ou otimizando suas soluções. A prática constante é a chave para a maestria.

Por último, mas não menos importante, compartilhe seu conhecimento. A comunidade de desenvolvedores é conhecida por sua colaboração e apoio mútuo. Ao ensinar outros, você solidifica ainda mais seu próprio aprendizado e contribui para o crescimento coletivo.

Obrigado por escolher este eBook como sua ferramenta de aprendizado. Esperamos que ele tenha sido valioso em sua jornada. Desejamos sucesso em todos os seus futuros empreendimentos de codificação e que você continue sendo uma força positiva e inovadora no mundo do desenvolvimento.