Javascript Avançado

Sumário

[1. Programação orientada a objetos em JavaScript 2](#_Toc131161127)

[1.1 Conceitos básicos de POO (encapsulamento, herança, polimorfismo) 3](#_Toc131161128)

[Encapsulamento: 3](#_Toc131161129)

[Herança: 3](#_Toc131161130)

[Polimorfismo: 3](#_Toc131161131)

[Exemplos de códigos: 3](#_Toc131161132)

[1.2 Entendendo como os protótipos funcionam em JavaScript 8](#_Toc131161133)

[1.3 Criando objetos e instâncias em JavaScript 11](#_Toc131161134)

[1.4 Usando o operador "new" para criar objetos 14](#_Toc131161135)

[1.5 Definindo propriedades e métodos em objetos 15](#_Toc131161136)

[1.6 Acessando e alterando propriedades e métodos de objetos 17](#_Toc131161137)

[1.7 Criando subclasses e herança prototípica em JavaScript 19](#_Toc131161138)

[1.8 Entendendo o "this" em JavaScript e como ele é usado em POO 23](#_Toc131161139)

[1.9 Usando funções construtoras para criar objetos 25](#_Toc131161140)

[1.10 Trabalhando com getters e setters em objetos 27](#_Toc131161141)

[1.11 Entendendo a diferença entre objetos literais e funções construtoras 29](#_Toc131161142)

[1.12 Implementando interfaces em JavaScript 31](#_Toc131161143)

[1.13 Usando o método "prototype" para adicionar propriedades e métodos em objetos 34](#_Toc131161144)

[1.14 Utilizando o padrão de módulo em POO 36](#_Toc131161145)

[2. Manipulação de eventos e call-backs 42](#_Toc131161146)

[Assíncrona e síncrona 43](#_Toc131161147)

[Forma assíncrona: 43](#_Toc131161148)

[Forma síncrona: 44](#_Toc131161149)

[Manipulação de eventos em um formulário 45](#_Toc131161150)

[Callbacks 46](#_Toc131161151)

[3. Trabalhando com Promises e Async/Await 48](#_Toc131161152)

[4. Programação Funcional em JavaScript 56](#_Toc131161153)

[Funções puras em JS 56](#_Toc131161154)

[Imutabilidade 58](#_Toc131161155)

[Composição de funções 61](#_Toc131161156)

[Recursão 63](#_Toc131161157)

[Mais sobre recursão na programação funcional em javascript 64](#_Toc131161158)

[High-order functions: 68](#_Toc131161159)

[Closures: 69](#_Toc131161160)

[Currying: 70](#_Toc131161161)

[5. Closures e escopo em JavaScript 77](#_Toc131161162)

[6. Tipos de dados avançados em JavaScript (Map, Set, WeakMap, WeakSet) 85](#_Toc131161163)

[7. Desenvolvimento de aplicações SPA (Single Page Application) 92](#_Toc131161164)

[8. Testes unitários e testes de integração em JavaScript 92](#_Toc131161165)

[9. Uso de módulos em JavaScript (CommonJS, AMD, ES6) 92](#_Toc131161166)

[10. Trabalhando com APIs RESTful em JavaScript 92](#_Toc131161167)

[11. Integração com serviços de terceiros (Google Maps, Facebook, Twitter, etc.) 92](#_Toc131161168)

[12. Manipulação de arquivos e dados locais em JavaScript 92](#_Toc131161169)

[13. Trabalhando com WebGL e gráficos 3D em JavaScript 92](#_Toc131161170)

[14. Desenvolvimento de jogos em JavaScript 92](#_Toc131161171)

[15. Uso de WebSockets e WebRTC em JavaScript 92](#_Toc131161172)

[16. Desenvolvimento de aplicações offline em JavaScript 92](#_Toc131161173)

[17. Otimização de performance em JavaScript 92](#_Toc131161174)

[18. Uso de bibliotecas e frameworks (jQuery, Angular, React, Vue, etc.) 93](#_Toc131161175)

[19. Desenvolvimento de extensões para navegadores em JavaScript 93](#_Toc131161176)

[20. Trabalhando com bancos de dados NoSQL em JavaScript 93](#_Toc131161177)

[21. Desenvolvimento de aplicações mobile com frameworks híbridos (React Native, Ionic, etc.) 93](#_Toc131161178)

[22. Segurança em aplicações web com JavaScript 93](#_Toc131161179)

[23. Desenvolvimento de chatbots em JavaScript 93](#_Toc131161180)

[24. Uso de tecnologias como WebAssembly em JavaScript 93](#_Toc131161181)

[25. Desenvolvimento de aplicações de realidade virtual e aumentada com JavaScript 93](#_Toc131161182)

[26. Trabalhando com Blockchain em JavaScript 93](#_Toc131161183)

[27. Uso de Machine Learning em JavaScript 94](#_Toc131161184)

[28. Desenvolvimento de aplicações de reconhecimento de voz em JavaScript 94](#_Toc131161185)

[29. Desenvolvimento de assistentes virtuais com JavaScript 94](#_Toc131161186)

[30. Desenvolvimento de aplicações para IoT com JavaScript 94](#_Toc131161187)

# Programação orientada a objetos em JavaScript

A programação orientada a objetos (POO) é um paradigma de programação que permite estruturar o código de forma mais organizada e modular, facilitando a manutenção e reutilização de código. Em JavaScript, a POO é implementada usando protótipos, em vez de classes como em outras linguagens de programação. Abaixo estão alguns tópicos que podem ser estudados para aprofundar a POO em JavaScript:

## Conceitos básicos de POO (encapsulamento, herança, polimorfismo)

JavaScript é uma linguagem de programação orientada a objetos, embora não siga uma abordagem de POO pura como outras linguagens, como Java e C++. No entanto, os conceitos básicos de POO (encapsulamento, herança e polimorfismo) também podem ser aplicados em JavaScript. Veja como esses conceitos são aplicados em JavaScript:

### Encapsulamento:

Em JavaScript, o encapsulamento é alcançado através do uso de funções construtoras e closures. As funções construtoras são usadas para criar objetos com propriedades privadas e públicas, enquanto closures são usados para proteger o acesso a essas propriedades privadas. Isso significa que os usuários externos só podem acessar as propriedades e métodos públicos do objeto e não podem modificar ou acessar diretamente as propriedades privadas.

### Herança:

Em JavaScript, a herança é alcançada usando protótipos. Os objetos em JavaScript têm um protótipo interno, que é um objeto do qual herdam propriedades e métodos. Uma classe pode ser definida como uma função construtora, que define as propriedades e métodos do objeto, e então pode ser usada como um protótipo para criar objetos derivados. A herança também pode ser alcançada usando a palavra-chave "extends" em classes ES6.

### Polimorfismo:

Em JavaScript, o polimorfismo é alcançado por meio da substituição de métodos em objetos derivados. Quando um objeto derivado tem um método com o mesmo nome e assinatura que um método na classe pai, ele substitui o método pai. Isso significa que o objeto derivado pode ter comportamentos diferentes para o mesmo método, permitindo que o código seja mais flexível e genérico.

Em resumo, os conceitos básicos de POO (encapsulamento, herança e polimorfismo) podem ser aplicados em JavaScript usando funções construtoras, closures, protótipos e palavras-chave como "extends".

#### Exemplos de códigos:

Exemplo de encapsulamento em JavaScript usando funções construtoras e closures:

*function* Car(*model*, *year*, *price*) {

*var* \_price = *price*; // propriedade privada

    this.model = *model*; // propriedade pública

    this.year = *year*; // propriedade pública

    // método público para acessar propriedade privada

    this.getPrice = *function* () {

        return \_price;

    };

    // método público para alterar propriedade privada

    this.setPrice = *function* (*newPrice*) {

        \_price = *newPrice*;

    };

}

*var* myCar = new Car("Honda Civic", 2022, 25000);

console.log(myCar.model); // "Honda Civic"

console.log(myCar.getPrice()); // 25000

myCar.setPrice(30000);

console.log(myCar.getPrice()); // 30000

Neste exemplo, a propriedade \_price é privada e só pode ser acessada ou modificada pelos métodos públicos getPrice e setPrice. As propriedades model e year são públicas e podem ser acessadas diretamente pelo objeto myCar.

Exemplo de herança em JavaScript usando protótipos:

*function* Animal(*name*, *sound*) {

    this.name = *name*;

    this.sound = *sound*;

}

Animal.prototype.makeSound = *function* () {

    console.log(this.sound);

};

*function* Cat(*name*) {

    Animal.call(this, *name*, "meow");

}

Cat.prototype = *Object*.create(Animal.prototype);

Cat.prototype.constructor = Cat;

*var* myCat = new Cat("Whiskers");

myCat.makeSound(); // "meow"

console.log(myCat instanceof Animal); // true

console.log(myCat instanceof Cat); // true

Neste exemplo, Animal é a classe pai que define a propriedade name e o método makeSound. Cat é a classe filha que herda name de Animal e adiciona o som "meow" como sua própria propriedade. O objeto myCat é uma instância de Cat que também herda o método makeSound de Animal.

Exemplo de polimorfismo em JavaScript usando a substituição de métodos:

*class* Shape {

    area() {

        return 0;

    }

}

*class* Rectangle extends Shape {

*constructor*(*width*, *height*) {

        super();

        this.width = *width*;

        this.height = *height*;

    }

    area() {

        return this.width \* this.height;

    }

}

*class* Circle extends Shape {

*constructor*(*radius*) {

        super();

        this.radius = *radius*;

    }

    area() {

        return Math.PI \* Math.pow(this.radius, 2);

    }

}

*let* shapes = [new Rectangle(10, 5), new Circle(7)];

shapes.forEach(*shape* *=>* console.log(*shape*.area()));

Neste exemplo, a classe Shape define o método area que retorna zero. Rectangle e Circle são classes filhas que substituem o método area de Shape com suas próprias implementações que calculam a área de um retângulo ou um círculo, respectivamente. O array shapes contém uma instância de cada classe e chama o método area em cada objeto, produzindo resultados diferentes com base no tipo de objeto. Isso é um exemplo de polimorfismo, onde o mesmo método pode ter comportamentos diferentes dependendo do tipo do objeto.

Aqui estão mais 3 exemplos de código em JavaScript aplicando encapsulamento, herança e polimorfismo:

Encapsulamento:

*function* Person(*name*, *age*) {

*let* privateAge = *age*; // propriedade privada

    this.getName = *function* () {

        return *name*; // propriedade pública

    };

    this.getAge = *function* () {

        return privateAge; // propriedade privada

    };

    this.setAge = *function* (*newAge*) {

        privateAge = *newAge*; // método que permite modificar a propriedade privada

    };

}

*const* person1 = new Person("João", 25);

console.log(person1.getName()); // "João"

console.log(person1.getAge()); // 25

person1.setAge(30);

console.log(person1.getAge()); // 30

Nesse exemplo, a função construtora Person cria um objeto com uma propriedade pública name e uma propriedade privada privateAge, que só pode ser acessada ou modificada por meio do método getAge e setAge. O uso de closures garante o encapsulamento da propriedade privada, impedindo que ela seja acessada diretamente por usuários externos.

Herança:

*function* Animal(*name*, *sound*) {

    this.name = *name*;

    this.sound = *sound*;

}

Animal.prototype.makeSound = *function* () {

    console.log(this.sound);

};

*function* Dog(*name*, *sound*) {

    Animal.call(this, *name*, *sound*); // chamada ao construtor da classe pai

}

Dog.prototype = *Object*.create(Animal.prototype); // estabelece a herança

Dog.prototype.constructor = Dog;

Dog.prototype.bark = *function* () {

    console.log("Woof!");

};

*const* dog1 = new Dog("Buddy", "au au");

dog1.makeSound(); // "au au"

dog1.bark(); // "Woof!"

Nesse exemplo, a classe Animal é definida como uma função construtora que cria um objeto com as propriedades name e sound e o método makeSound. A classe Dog herda as propriedades e métodos da classe Animal usando protótipos e adiciona seu próprio método bark. Quando um objeto Dog é criado, ele tem acesso aos métodos de sua classe pai e aos seus próprios métodos.

Polimorfismo:

*class* Shape {

*constructor*(*width*, *height*) {

        this.width = *width*;

        this.height = *height*;

    }

    getArea() {

        return this.width \* this.height;

    }

}

*class* Square extends Shape {

*constructor*(*side*) {

        super(*side*, *side*);

    }

    getArea() {

        return super.getArea();

    }

}

*class* Rectangle extends Shape {

*constructor*(*width*, *height*) {

        super(*width*, *height*);

    }

    getArea() {

        return super.getArea();

    }

}

*const* square1 = new Square(5);

console.log(square1.getArea()); // 25

*const* rectangle1 = new Rectangle(4, 6);

console.log(rectangle1.getArea()); // 24

Nesse exemplo, a classe Shape é definida como uma classe base para outras formas geométricas e possui um método getArea que retorna a área da forma. As classes Square e Rectangle são derivadas da classe Shape e substituem o método getArea para calcular a área de cada forma de maneira polimórfica, podemos definir um método area() na classe Forma que será substituído nas subclasses. Aqui está o código completo:

*class* Forma {

*constructor*(*cor*) {

        this.cor = *cor*;

    }

    area() {

        throw new *Error*('Método area() não implementado.');

    }

}

*class* Retangulo extends Forma {

*constructor*(*cor*, *largura*, *altura*) {

        super(*cor*);

        this.largura = *largura*;

        this.altura = *altura*;

    }

    area() {

        return this.largura \* this.altura;

    }

}

*class* Circulo extends Forma {

*constructor*(*cor*, *raio*) {

        super(*cor*);

        this.raio = *raio*;

    }

    area() {

        return Math.PI \* this.raio \*\* 2;

    }

}

*const* retangulo = new Retangulo('verde', 5, 10);

*const* circulo = new Circulo('vermelho', 7);

console.log(retangulo.area()); // 50

console.log(circulo.area()); // 153.93804002589985

Neste exemplo, criamos uma classe Forma que define um método area() que é lançado como uma exceção se não for implementado nas subclasses. Em seguida, criamos duas subclasses Retangulo e Circulo que implementam o método area() para calcular a área da forma correspondente. Por fim, criamos objetos retangulo e circulo e chamamos seus métodos area(), que retornam os resultados corretos.

## Entendendo como os protótipos funcionam em JavaScript

Em JavaScript, os protótipos são usados para criar herança entre objetos. Um protótipo é um objeto a partir do qual outros objetos herdam propriedades e métodos. Cada objeto em JavaScript tem um protótipo que pode ser acessado através da propriedade \_\_proto\_\_.

Quando uma propriedade ou método é procurado em um objeto, o mecanismo de busca começa pelo próprio objeto e depois procura em seu protótipo e assim por diante até encontrar a propriedade ou método desejado ou chegar ao topo da cadeia de protótipos (que normalmente é o objeto Object.prototype).

Por exemplo, vamos criar um objeto Pessoa e adicionar uma propriedade nome e um método dizerNome ao seu protótipo:

// Criando o objeto Pessoa

*var* Pessoa = *function* (*nome*) {

    this.nome = *nome*;

};

// Adicionando um método ao protótipo de Pessoa

Pessoa.prototype.dizerNome = *function* () {

    console.log('Meu nome é ' + this.nome);

};

// Criando um objeto a partir de Pessoa

*var* pessoa1 = new Pessoa('João');

// Chamando o método dizerNome do objeto pessoa1

pessoa1.dizerNome(); // Saída: Meu nome é João

Nesse exemplo, quando chamamos o método dizerNome no objeto pessoa1, o mecanismo de busca começa procurando no próprio objeto pessoa1. Como dizerNome não é encontrado nesse objeto, o mecanismo procura no protótipo de pessoa1, que é Pessoa.prototype. Lá, ele encontra o método dizerNome e o executa.

Os protótipos são uma parte importante do sistema de herança em JavaScript e são usados extensivamente em bibliotecas e frameworks como o jQuery e o AngularJS. É importante entender como eles funcionam para poder escrever código JavaScript eficiente e elegante.

Mais exemplos:

Exemplo 1: Adicionando métodos ao protótipo

// Criando um objeto Pessoa

*var* Pessoa = *function* (*nome*) {

    this.nome = *nome*;

};

// Adicionando um método ao protótipo de Pessoa

Pessoa.prototype.dizerOla = *function* () {

    console.log('Olá, meu nome é ' + this.nome);

};

// Criando um objeto a partir de Pessoa

*var* pessoa1 = new Pessoa('João');

// Chamando o método dizerOla do objeto pessoa1

pessoa1.dizerOla(); // Saída: Olá, meu nome é João

Nesse exemplo, adicionamos um método chamado dizerOla ao protótipo do objeto Pessoa. Esse método usa a propriedade nome do objeto para imprimir uma mensagem na tela. Quando chamamos o método dizerOla no objeto pessoa1, o mecanismo de busca encontra o método no protótipo de Pessoa e o executa.

Exemplo 2: Herança de protótipos

// Criando um objeto Animal

*var* Animal = *function*() {};

// Adicionando um método ao protótipo de Animal

Animal.prototype.andar = *function*() {

  console.log('O animal está andando');

};

// Criando um objeto Cachorro que herda de Animal

*var* Cachorro = *function*() {};

Cachorro.prototype = *Object*.create(Animal.prototype);

// Adicionando um método ao protótipo de Cachorro

Cachorro.prototype.latir = *function*() {

  console.log('O cachorro está latindo');

};

// Criando um objeto a partir de Cachorro

*var* cachorro1 = new Cachorro();

// Chamando o método andar do objeto cachorro1

cachorro1.andar(); // Saída: O animal está andando

// Chamando o método latir do objeto cachorro1

cachorro1.latir(); // Saída: O cachorro está latindo

Nesse exemplo, criamos dois objetos: Animal e Cachorro. Adicionamos um método chamado andar ao protótipo do objeto Animal. Em seguida, criamos um objeto Cachorro que herda do protótipo de Animal. Adicionamos um método chamado latir ao protótipo de Cachorro.

Quando chamamos o método andar no objeto cachorro1, o mecanismo de busca começa procurando no próprio objeto, mas como esse método não é encontrado, ele procura no protótipo do objeto, que é o protótipo de Cachorro (Animal.prototype). Lá, ele encontra o método andar e o executa.

Quando chamamos o método latir no objeto cachorro1, o mecanismo de busca começa procurando no próprio objeto, e como esse método é encontrado, ele é executado.

## Criando objetos e instâncias em JavaScript

Em JavaScript, podemos criar objetos e instâncias de objetos de diversas maneiras. Abaixo estão alguns exemplos:

Exemplo 1: Criando um objeto usando a notação literal

// Criando um objeto usando a notação literal

*var* pessoa = {

    nome: 'João',

    idade: 30,

    dizerOla: *function* () {

        console.log('Olá, meu nome é ' + this.nome);

    }

};

// Chamando o método dizerOla do objeto pessoa

pessoa.dizerOla(); // Saída: Olá, meu nome é João

Nesse exemplo, criamos um objeto chamado pessoa usando a notação literal. Esse objeto tem três propriedades: nome, idade e dizerOla, que é um método que usa a propriedade nome do objeto para imprimir uma mensagem na tela.

Exemplo 2: Criando um objeto usando uma função construtora

// Criando uma função construtora Pessoa

*var* Pessoa = *function*(*nome*, *idade*) {

    this.nome = *nome*;

    this.idade = *idade*;

    this.dizerOla = *function*() {

      console.log('Olá, meu nome é ' + this.nome);

    };

  };

  // Criando uma instância do objeto Pessoa

*var* pessoa1 = new Pessoa('João', 30);

  // Chamando o método dizerOla do objeto pessoa1

  pessoa1.dizerOla(); // Saída: Olá, meu nome é João

Nesse exemplo, criamos uma função construtora chamada Pessoa. Essa função tem dois parâmetros, nome e idade, e usa o operador this para criar duas propriedades no objeto criado a partir da função, nome e idade. Também cria um método chamado dizerOla que imprime uma mensagem na tela.

Depois, criamos uma instância do objeto Pessoa chamada pessoa1 usando a palavra-chave new. Podemos chamar o método dizerOla nessa instância para imprimir a mensagem na tela.

Exemplo 3: Criando um objeto usando o método Object.create()

// Criando um objeto Animal

*var* animal = {

    andar: *function* () {

        console.log('O animal está andando');

    }

};

// Criando um objeto Cachorro que herda do objeto Animal

*var* cachorro = *Object*.create(animal);

cachorro.latir = *function* () {

    console.log('O cachorro está latindo');

};

// Chamando o método andar do objeto cachorro

cachorro.andar(); // Saída: O animal está andando

// Chamando o método latir do objeto cachorro

cachorro.latir(); // Saída: O cachorro está latindo

Nesse exemplo, criamos um objeto chamado animal que tem um método chamado andar. Em seguida, criamos um objeto chamado cachorro usando o método Object.create(). Esse objeto herda do objeto animal, por isso pode acessar o método andar.

Adicionamos um método chamado latir ao objeto cachorro. Quando chamamos o método andar no objeto cachorro, o mecanismo de busca começa procurando no próprio objeto, mas como esse método não é encontrado, ele procura no protótipo do objeto, que é o objeto animal. L

Ao procurar uma propriedade ou método em um objeto, o mecanismo de busca começa procurando no próprio objeto. Se a propriedade ou método não for encontrado, o mecanismo de busca procura no protótipo do objeto. O protótipo é um objeto do qual o objeto atual herda propriedades e métodos.

No exemplo anterior, o objeto cachorro foi criado usando o método Object.create(), e seu protótipo foi definido como o objeto animal. Isso significa que o objeto cachorro herda todas as propriedades e métodos do objeto animal. Quando chamamos o método andar no objeto cachorro, o mecanismo de busca procura primeiro no próprio objeto cachorro, mas como esse método não é encontrado, ele procura no protótipo do objeto, que é o objeto animal. O método andar é encontrado no objeto animal, e é executado com sucesso.

O objeto cachorro também tem uma propriedade chamada latir, que não é herdada do objeto animal. Quando chamamos o método latir no objeto cachorro, o mecanismo de busca procura primeiro no próprio objeto cachorro, e encontra o método latir, que é executado com sucesso.

Essa é uma maneira de criar objetos com base em outros objetos, e pode ser muito útil para criar hierarquias de objetos e compartilhar código comum entre eles.

Exemplo 3: Criando um objeto usando uma função construtora

// Criando uma função construtora para criar objetos de Carro

*function* Carro(*marca*, *modelo*, *ano*) {

    this.marca = *marca*;

    this.modelo = *modelo*;

    this.ano = *ano*;

}

// Adicionando um método ao protótipo do objeto Carro

Carro.prototype.ligar = *function* () {

    console.log(`O ${this.modelo} está ligado.`);

}

// Criando uma instância do objeto Carro

*let* meuCarro = new Carro("Toyota", "Corolla", 2022);

// Acessando as propriedades e métodos do objeto Carro

console.log(meuCarro.marca); // "Toyota"

console.log(meuCarro.modelo); // "Corolla"

console.log(meuCarro.ano); // 2022

meuCarro.ligar(); // "O Corolla está ligado."

Neste exemplo, criamos um objeto Carro usando uma função construtora que recebe três parâmetros (marca, modelo, ano) e os atribui às propriedades correspondentes do objeto usando a palavra-chave this.

Em seguida, adicionamos um método ligar() ao protótipo do objeto Carro usando a propriedade prototype.

Por fim, criamos uma instância do objeto Carro chamada meuCarro, passando valores para os parâmetros da função construtora. Acessamos as propriedades e o método do objeto meuCarro usando a sintaxe de ponto.

Exemplo 4: Criando um objeto a partir de uma requisição em uma API

// Fazendo uma requisição a uma API

fetch('https://jsonplaceholder.typicode.com/todos/1')

    .then(*response* *=>* *response*.json())

    .then(*data* *=>* {

        // Criando um objeto Todo com os dados da API

*let* todo = {

            id: *data*.id,

            title: *data*.title,

            completed: *data*.completed

        };

        // Acessando as propriedades do objeto Todo

        console.log(todo.id); // 1

        console.log(todo.title); // "delectus aut autem"

        console.log(todo.completed); // false

    });

Neste exemplo, fazemos uma requisição a uma API usando o método fetch(). Em seguida, usamos o método json() para obter os dados da resposta da API no formato JSON.

Com esses dados, criamos um objeto Todo com três propriedades (id, title, completed) e atribuímos os valores obtidos da API a essas propriedades.

Finalmente, acessamos as propriedades do objeto Todo usando a sintaxe de ponto e imprimimos seus valores no console.

## Usando o operador "new" para criar objetos

O operador new é usado em JavaScript para criar uma nova instância de um objeto. Quando usamos o operador new com uma função construtora, a função é chamada e retorna um novo objeto. O objeto retornado é então atribuído à variável que está recebendo a instância.

Aqui está um exemplo simples de como usar o operador new para criar objetos:

// Criando uma função construtora para criar objetos Pessoa

*function* Pessoa(*nome*, *idade*) {

    this.nome = *nome*;

    this.idade = *idade*;

}

// Criando uma nova instância do objeto Pessoa usando o operador new

*let* pessoa1 = new Pessoa("João", 25);

*let* pessoa2 = new Pessoa("Maria", 30);

// Acessando as propriedades da instância do objeto Pessoa

console.log(pessoa1.nome); // "João"

console.log(pessoa1.idade); // 25

console.log(pessoa2.nome); // "Maria"

console.log(pessoa2.idade); // 30

Exemplo 2: Criando um objeto para calcular a velocidade final de um objeto em queda livre

// Definindo uma função construtora para criar objetos QuedaLivre

*function* QuedaLivre(*altura*) {

    this.altura = *altura*;

    this.velocidadeInicial = 0;

    this.velocidadeFinal = *function*() {

      return Math.sqrt(2 \* 9.8 \* this.altura);

    };

  }

  // Criando uma nova instância do objeto QuedaLivre usando o operador new

*let* quedaLivre1 = new QuedaLivre(10);

  // Acessando as propriedades e o método do objeto QuedaLivre

  console.log(quedaLivre1.altura); // 10

  console.log(quedaLivre1.velocidadeInicial); // 0

  console.log(quedaLivre1.velocidadeFinal()); // 14

Neste exemplo, definimos uma função construtora chamada QuedaLivre que aceita um parâmetro (altura) e atribui esse valor à propriedade correspondente do objeto usando a palavra-chave this.

Além disso, adicionamos um método velocidadeFinal() ao objeto QuedaLivre que calcula a velocidade final de um objeto em queda livre usando a fórmula vf = sqrt(2 \* g \* h), onde g é a aceleração da gravidade (assumindo um valor padrão de 9.8 m/s^2) e h é a altura do objeto em queda livre.

Em seguida, usamos o operador new para criar uma nova instância do objeto QuedaLivre chamada quedaLivre1, passando um valor para o parâmetro da função construtora.

Por fim, acessamos as propriedades e o método do objeto quedaLivre1 usando a sintaxe de ponto e imprimimos seus valores no console.

## Definindo propriedades e métodos em objetos

Em JavaScript, você pode definir propriedades e métodos em objetos usando a sintaxe de literal de objeto ({}) ou usando uma função construtora.

Usando a sintaxe de literal de objeto, você pode definir propriedades e métodos da seguinte maneira:

// Criando um objeto pessoa com propriedades e métodos

*let* pessoa = {

    nome: "João",

    idade: 30,

    profissao: "Engenheiro",

    cumprimentar: *function* () {

        console.log("Olá, meu nome é " + this.nome);

    }

};

// Acessando as propriedades e o método do objeto pessoa

console.log(pessoa.nome); // "João"

console.log(pessoa.idade); // 30

console.log(pessoa.profissao); // "Engenheiro"

pessoa.cumprimentar(); // "Olá, meu nome é João"

Neste exemplo, criamos um objeto pessoa com propriedades nome, idade e profissao, e um método cumprimentar() que imprime uma mensagem de saudação no console usando a propriedade nome do objeto.

Usando uma função construtora, você pode definir propriedades e métodos da seguinte maneira:

// Definindo uma função construtora para criar objetos Carro

*function* Carro(*modelo*, *ano*, *cor*) {

    this.modelo = *modelo*;

    this.ano = *ano*;

    this.cor = *cor*;

    this.descricao = *function* () {

        console.log("Este carro é um " + this.modelo + " " + this.cor + " fabricado em " + this.ano);

    };

}

// Criando uma nova instância do objeto Carro usando o operador new

*let* carro1 = new Carro("Fusca", 1970, "vermelho");

// Acessando as propriedades e o método do objeto Carro

console.log(carro1.modelo); // "Fusca"

console.log(carro1.ano); // 1970

console.log(carro1.cor); // "vermelho"

carro1.descricao(); // "Este carro é um Fusca vermelho fabricado em 1970"

Neste exemplo, definimos uma função construtora Carro que aceita três parâmetros (modelo, ano e cor) e atribui esses valores às propriedades correspondentes do objeto usando a palavra-chave this. Além disso, adicionamos um método descricao() ao objeto Carro que imprime uma mensagem descrevendo o carro no console.

Em seguida, usamos o operador new para criar uma nova instância do objeto Carro chamada carro1, passando valores para os parâmetros da função construtora.

Por fim, acessamos as propriedades e o método do objeto carro1 usando a sintaxe de ponto e imprimimos seus valores no console.

Como outro exemplo de objeto em JavaScript usando agora conceitos de **ethical hacking**, podemos criar um objeto Hacker com propriedades e métodos que simulem as ações de um hacker ético.

// Criando um objeto Hacker com propriedades e métodos

*let* Hacker = {

    nome: "Alice",

    profissao: "Hacker Ético",

    habilidades: ["Programação", "Pentest", "Engenharia Reversa"],

    invadir: *function* (*alvo*) {

        console.log("Invasão em andamento...");

        // Lógica de invasão ao alvo

        console.log(`Invasão concluída. Dados do alvo ${*alvo*} obtidos.`);

    },

    reportar: *function* (*alvo*, *vulnerabilidades*) {

        console.log(`Foram encontradas as seguintes vulnerabilidades no alvo ${*alvo*}: ${*vulnerabilidades*}`);

        // Lógica de reporte das vulnerabilidades encontradas

        console.log(`Relatório enviado para a equipe de segurança do alvo ${*alvo*}.`);

    }

};

// Acessando as propriedades e o método do objeto Hacker

console.log(Hacker.nome); // "Alice"

console.log(Hacker.profissao); // "Hacker Ético"

console.log(Hacker.habilidades); // ["Programação", "Pentest", "Engenharia Reversa"]

Hacker.invadir("Empresa X"); // "Invasão em andamento... Invasão concluída. Dados do alvo Empresa X obtidos."

Hacker.reportar("Empresa X", "SQL Injection, XSS"); // "Foram encontradas as seguintes vulnerabilidades no alvo Empresa X: SQL Injection, XSS. Relatório enviado para a equipe de segurança do alvo Empresa X."

Neste exemplo, criamos um objeto Hacker com propriedades nome, profissao e habilidades, e dois métodos invadir() e reportar() que simulam as ações de um hacker ético.

O método invadir() recebe como parâmetro o nome do alvo que o hacker deseja invadir e imprime uma mensagem simulando a invasão. A lógica de invasão real é deixada para o exemplo.

O método reportar() recebe como parâmetros o nome do alvo e as vulnerabilidades encontradas durante a invasão, e imprime uma mensagem simulando o reporte das vulnerabilidades encontradas. A lógica de reporte real também é deixada para o exemplo.

## Acessando e alterando propriedades e métodos de objetos

Para acessar e alterar propriedades e métodos de objetos em JavaScript, podemos utilizar a sintaxe de ponto (.) ou de colchetes ([]). Vamos criar um objeto pessoa e demonstrar como acessar e alterar suas propriedades e métodos:

// Criando um objeto pessoa

*let* pessoa = {

    nome: "João",

    idade: 30,

    profissao: "Programador",

    saudacao: *function* () {

        console.log(`Olá, meu nome é ${this.nome}, tenho ${this.idade} anos e sou ${this.profissao}.`);

    }

};

// Acessando as propriedades e o método do objeto pessoa

console.log(pessoa.nome); // "João"

console.log(pessoa["idade"]); // 30

pessoa.saudacao(); // "Olá, meu nome é João, tenho 30 anos e sou Programador."

// Alterando as propriedades do objeto pessoa

pessoa.nome = "Maria";

pessoa["idade"] = 35;

console.log(pessoa.nome); // "Maria"

console.log(pessoa.idade); // 35

// Adicionando uma nova propriedade ao objeto pessoa

pessoa.salario = 5000;

console.log(pessoa.salario); // 5000

// Alterando o método do objeto pessoa

pessoa.saudacao = *function* () {

    console.log(`Olá, meu nome é ${this.nome}, tenho ${this.idade} anos e trabalho como ${this.profissao}.`);

}

pessoa.saudacao(); // "Olá, meu nome é Maria, tenho 35 anos e trabalho como Programador."

No exemplo acima, criamos um objeto pessoa com propriedades nome, idade, profissao e um método saudacao(), que imprime uma mensagem de saudação com as informações da pessoa.

Usando a sintaxe de ponto e de colchetes, acessamos as propriedades do objeto pessoa, como nome e idade. Para acessar o método saudacao(), usamos a sintaxe de ponto seguida de parênteses.

Para alterar as propriedades do objeto pessoa, usamos novamente as sintaxes de ponto e de colchetes, atribuindo novos valores às propriedades existentes. Também adicionamos uma nova propriedade salario ao objeto pessoa.

Para alterar o método saudacao() do objeto pessoa, atribuímos uma nova função a ele, usando a sintaxe de ponto. Note que, dentro da nova função, usamos a palavra-chave this para referenciar as propriedades do objeto atual (pessoa).

Vamos criar um objeto **particula** para representar uma partícula quântica e demonstrar como acessar e alterar suas propriedades e métodos:

// Criando um objeto particula

*let* particula = {

    posicao: { x: 0, y: 0, z: 0 },

    momento: { px: 0, py: 0, pz: 0 },

    spin: 0.5,

    energia: *function* () {

        return Math.sqrt(Math.pow(this.momento.px, 2) + Math.pow(this.momento.py, 2) + Math.pow(this.momento.pz, 2));

    }

};

// Acessando as propriedades e o método do objeto particula

console.log(particula.posicao.x); // 0

console.log(particula["momento"]["pz"]); // 0

console.log(particula.energia()); // 0

// Alterando as propriedades do objeto particula

particula.posicao.x = 1;

particula["momento"]["pz"] = 2;

console.log(particula.posicao.x); // 1

console.log(particula.momento.pz); // 2

// Alterando o método do objeto particula

particula.energia = *function* () {

    return Math.sqrt(Math.pow(this.momento.px, 2) + Math.pow(this.momento.py, 2) + Math.pow(this.momento.pz, 2)) + this.spin \* 2;

}

console.log(particula.energia()); // 2.5

No exemplo acima, criamos um objeto particula com propriedades posicao, momento, spin e um método energia(), que calcula a energia da partícula com base no momento e no spin.

Usando a sintaxe de ponto e de colchetes, acessamos as propriedades do objeto particula, como posicao e momento. Para acessar o método energia(), usamos a sintaxe de ponto seguida de parênteses.

Para alterar as propriedades do objeto particula, usamos novamente as sintaxes de ponto e de colchetes, atribuindo novos valores às propriedades existentes.

Para alterar o método energia() do objeto particula, atribuímos uma nova função a ele, usando a sintaxe de ponto. A nova função adiciona uma correção no cálculo da energia, levando em conta o spin da partícula. Note que, dentro da nova função, usamos a palavra-chave this para referenciar as propriedades do objeto atual (particula).

## Criando subclasses e herança prototípica em JavaScript

Vamos criar algumas subclasses para representar diferentes tipos de plantas no Reino Vegetal:

// Definindo uma classe base para plantas

*class* Planta {

*constructor*(*nome*, *cor*, *altura*) {

        this.nome = *nome*;

        this.cor = *cor*;

        this.altura = *altura*;

    }

    crescer() {

        console.log(`${this.nome} está crescendo...`);

        this.altura += 10;

    }

    florescer() {

        console.log(`${this.nome} está florescendo com cor ${this.cor}...`);

    }

}

// Definindo uma subclasse para árvores

*class* Arvore extends Planta {

*constructor*(*nome*, *cor*, *altura*, *tronco*) {

        super(*nome*, *cor*, *altura*);

        this.tronco = *tronco*;

    }

    cairFolhas() {

        console.log(`${this.nome} está perdendo as folhas do tronco ${this.tronco}...`);

    }

}

// Definindo uma subclasse para flores

*class* Flor extends Planta {

*constructor*(*nome*, *cor*, *altura*, *cheiro*) {

        super(*nome*, *cor*, *altura*);

        this.cheiro = *cheiro*;

    }

    exalarCheiro() {

        console.log(`${this.nome} está exalando o cheiro ${this.cheiro}...`);

    }

}

// Criando objetos das subclasses

*let* carvalho = new Arvore("Carvalho", "marrom", 200, "principal");

carvalho.florescer(); // Carvalho está florescendo com cor marrom...

carvalho.cairFolhas(); // Carvalho está perdendo as folhas do tronco principal...

console.log(carvalho.altura); // 200

carvalho.crescer();

console.log(carvalho.altura); // 210

*let* margarida = new Flor("Margarida", "branco", 30, "doce");

margarida.florescer(); // Margarida está florescendo com cor branco...

margarida.exalarCheiro(); // Margarida está exalando o cheiro doce...

console.log(margarida.altura); // 30

margarida.crescer();

console.log(margarida.altura); // 40

No exemplo acima, definimos uma classe base Planta, que tem propriedades como nome, cor e altura, e métodos como crescer() e florescer(). Em seguida, definimos duas subclasses: Arvore e Flor, que herdam da classe base Planta usando a palavra-chave extends.

A classe Arvore adiciona uma nova propriedade tronco e um novo método cairFolhas(), que não existem na classe base. Já a classe Flor adiciona uma nova propriedade cheiro e um novo método exalarCheiro().

Usando a palavra-chave super, podemos chamar o construtor da classe base a partir das subclasses, passando os argumentos necessários para as propriedades da classe base. Também podemos acessar os métodos da classe base usando a sintaxe super.metodo().

Para criar objetos das subclasses, basta instanciar as classes usando o operador new, passando os argumentos necessários para as propriedades das subclasses. Podemos acessar as propriedades e métodos das subclasses usando as sintaxes de ponto e de objeto, da mesma forma que fizemos com a classe base.

No exemplo acima, criamos dois objetos: carvalho e margarida, que são instâncias das subclasses Arvore e Flor, respectivamente. Podemos acessar as propriedades e métodos de cada objeto usando a sintaxe de ponto:

console.log(carvalho.nome); // Carvalho

console.log(margarida.cor); // branco

Também podemos acessar as propriedades e métodos da classe base a partir das subclasses:

console.log(carvalho.altura); // 200

console.log(margarida.florescer()); // Margarida está florescendo com cor branco...

E, como vimos anteriormente, podemos alterar o comportamento de métodos da classe base nas subclasses, sobrescrevendo-os com novas implementações:

*class* Arvore extends *Planta* {

    // ...

    florescer() {

        console.log(`${this.nome} não tem flores, apenas frutas...`);

    }

}

*let* laranjeira = new Arvore("Laranjeira", "verde", 300, "principal");

laranjeira.florescer(); // Laranjeira não tem flores, apenas frutas...

Nesse exemplo, a classe Arvore sobrescreve o método florescer() da classe base Planta para informar que a árvore não tem flores, apenas frutas.

Em resumo, as subclasses permitem estender a funcionalidade de uma classe base, adicionando novas propriedades e métodos ou sobrescrevendo os existentes. Isso nos permite criar uma hierarquia de classes mais complexas e adaptáveis às nossas necessidades.

Vamos criar outro exemplo de subclasses e herança prototípica usando o Reino dos Fungos como tema. Vamos supor que temos uma classe base Fungo que contém as propriedades comuns a todos os fungos, como nome científico, cor, habitat e tipo de nutrição. Em seguida, criaremos duas subclasses Cogumelo e Champignon que herdam as propriedades da classe base e adicionam suas próprias propriedades e métodos específicos.

*class* Fungo {

*constructor*(*nomeCientifico*, *cor*, *habitat*, *nutricao*) {

        this.nomeCientifico = *nomeCientifico*;

        this.cor = *cor*;

        this.habitat = *habitat*;

        this.nutricao = *nutricao*;

    }

    crescer() {

        console.log(`${this.nomeCientifico} está crescendo...`);

    }

    reproduzir() {

        console.log(`${this.nomeCientifico} está se reproduzindo...`);

    }

}

*class* Cogumelo extends Fungo {

*constructor*(*nomeCientifico*, *cor*, *habitat*, *nutricao*, *venenoso*) {

        super(*nomeCientifico*, *cor*, *habitat*, *nutricao*);

        this.venenoso = *venenoso*;

    }

    soltarEsporos() {

        console.log(`${this.nomeCientifico} está soltando seus esporos...`);

    }

    comer() {

        if (this.venenoso) {

            console.log(`${this.nomeCientifico} é venenoso! Não coma!`);

        } else {

            console.log(`${this.nomeCientifico} é comestível. Bom apetite!`);

        }

    }

}

*class* Champignon extends Fungo {

*constructor*(*nomeCientifico*, *cor*, *habitat*, *nutricao*, *sabor*) {

        super(*nomeCientifico*, *cor*, *habitat*, *nutricao*);

        this.sabor = *sabor*;

    }

    cozinhar() {

        console.log(`${this.nomeCientifico} está sendo preparado para o jantar...`);

    }

    sabor() {

        console.log(`${this.nomeCientifico} tem sabor ${this.sabor}.`);

    }

}

Nesse exemplo, a classe Cogumelo tem uma propriedade adicional venenoso que indica se o cogumelo é venenoso ou comestível, e dois métodos adicionais soltarEsporos() e comer(). A classe Champignon tem uma propriedade adicional sabor que indica o sabor do cogumelo e dois métodos adicionais cozinhar() e sabor().

Podemos criar instâncias dessas subclasses da mesma maneira que criamos instâncias da classe base Fungo:

*let* amanita = new Cogumelo("Amanita muscaria", "vermelho e branco", "floresta", "saprófita", true);

*let* shitake = new Champignon("Lentinula edodes", "marrom", "floresta", "saprófita", "umami");

Podemos acessar as propriedades e métodos desses objetos da mesma forma que fizemos no exemplo anterior:

console.log(amanita.nomeCientifico); // Amanita muscaria

console.log(shitake.sabor); // umami

amanita.comer();

## Entendendo o "this" em JavaScript e como ele é usado em POO

Em JavaScript, a palavra-chave this é usada para se referir ao objeto atual em que um determinado código está sendo executado. O this é uma propriedade dinâmica que pode ser diferente em cada chamada de uma função ou método, dependendo do contexto de execução em que se encontra.

O uso do this é especialmente importante em Programação Orientada a Objetos (POO), pois permite que os métodos das classes se refiram ao objeto atual em que estão sendo executados. O this é geralmente usado dentro dos métodos de uma classe para se referir a propriedades ou métodos específicos daquele objeto.

Por exemplo, suponha que temos a classe Person com as propriedades name e age, e o método greet que imprime uma mensagem de saudação na tela com o nome da pessoa:

*class* Person {

*constructor*(*name*, *age*) {

        this.name = *name*;

        this.age = *age*;

    }

    greet() {

        console.log(`Hello, my name is ${this.name}`);

    }

}

Ao instanciar um objeto person1 da classe Person, podemos chamar o método greet:

*const* person1 = new Person('John', 30);

person1.greet(); // Output: "Hello, my name is John"

Nesse caso, quando o método greet é chamado em person1, o this se refere ao próprio objeto person1. Dessa forma, podemos acessar as propriedades name e age daquele objeto dentro do método.

O uso correto do this é importante para evitar erros e bugs no código. Por exemplo, se o this for usado fora de um método de objeto ou de uma classe, ele se referirá ao objeto global window no caso de navegadores ou ao objeto global no caso do Node.js.

Outra situação comum em que o uso do this pode causar problemas é quando uma função é chamada fora do contexto em que foi definida. Nesse caso, o this pode se referir a um objeto inesperado ou gerar um erro de referência. Para contornar esse problema, podemos usar as funções bind, call e apply para definir explicitamente o objeto this que será usado em uma determinada chamada de função.

Vamos criar uma classe Outfit (look) que representa um conjunto de roupas para uma ocasião especial na moda executiva, e uma subclasse BusinessSuit (ternos executivos) que herda as propriedades e métodos da classe Outfit, mas adiciona algumas propriedades específicas para ternos executivos.

*class* Outfit {

*constructor*(*shirt*, *pants*, *shoes*) {

        this.shirt = *shirt*;

        this.pants = *pants*;

        this.shoes = *shoes*;

    }

    describe() {

        console.log(`This outfit consists of a ${this.shirt}, ${this.pants}, and ${this.shoes}.`);

    }

}

*class* BusinessSuit extends Outfit {

*constructor*(*shirt*, *pants*, *shoes*, *tie*, *jacket*) {

        super(*shirt*, *pants*, *shoes*);

        this.tie = *tie*;

        this.jacket = *jacket*;

    }

    describe() {

        console.log(`This business suit consists of a ${this.shirt}, ${this.pants}, ${this.tie}, ${this.jacket}, and ${this.shoes}.`);

    }

    showOff() {

        console.log(`Check out my professional look! I'm wearing a ${this.jacket} and ${this.tie}.`);

    }

}

Nesse exemplo, a classe Outfit possui as propriedades shirt (camisa), pants (calça) e shoes (sapatos) e um método describe para imprimir uma descrição básica do conjunto de roupas. A subclasse BusinessSuit estende a classe Outfit e adiciona as propriedades tie (gravata) e jacket (paletó) e o método showOff para exibir uma mensagem específica para um terno executivo.

Podemos criar um objeto suit1 da classe BusinessSuit e chamar os métodos describe e showOff:

*const* suit1 = new BusinessSuit('white shirt', 'black pants', 'black shoes', 'red tie', 'navy jacket');

suit1.describe(); // Output: "This business suit consists of a white shirt, black pants, red tie, navy jacket, and black shoes."

suit1.showOff(); // Output: "Check out my professional look! I'm wearing a navy jacket and red tie."

Nesse caso, quando os métodos describe e showOff são chamados em suit1, o this se refere ao próprio objeto suit1, que é uma instância da classe BusinessSuit. Dessa forma, podemos acessar as propriedades específicas da subclasse BusinessSuit, como tie e jacket, dentro do método showOff.

## Usando funções construtoras para criar objetos

Em JavaScript, também é possível criar objetos usando funções construtoras. Uma função construtora é uma função que pode ser usada para criar objetos. Para criar um objeto usando uma função construtora, usamos a palavra-chave new seguida do nome da função construtora.

Aqui está um exemplo usando o tema de receitas de cozinha:

*function* Recipe(*title*, *ingredients*, *instructions*) {

    this.title = *title*;

    this.ingredients = *ingredients*;

    this.instructions = *instructions*;

    this.printIngredients = *function* () {

        console.log(`Ingredients for ${this.title}:`);

        for (*let* ingredient of this.ingredients) {

            console.log(`- ${ingredient}`);

        }

    };

}

*const* chocolateCake = new Recipe(

    "Chocolate Cake",

    ["flour", "sugar", "cocoa powder", "baking powder", "eggs", "milk", "vegetable oil"],

    "1. Preheat oven to 350°F. 2. Combine all ingredients in a mixing bowl. 3. Beat with an electric mixer until smooth. 4. Pour into a greased baking dish. 5. Bake for 35-40 minutes or until a toothpick inserted in the center comes out clean."

);

chocolateCake.printIngredients(); // Output: "Ingredients for Chocolate Cake: - flour - sugar - cocoa powder - baking powder - eggs - milk - vegetable oil"

Nesse exemplo, temos a função construtora Recipe, que recebe três parâmetros: title (título), ingredients (ingredientes) e instructions (instruções). Dentro da função construtora, definimos as propriedades title, ingredients e instructions usando a palavra-chave this. Também definimos um método printIngredients que imprime os ingredientes da receita na saída do console.

Para criar um objeto usando a função construtora Recipe, usamos a palavra-chave new seguida do nome da função e passamos os parâmetros necessários. No exemplo acima, criamos o objeto chocolateCake usando a função construtora Recipe e passamos o título, ingredientes e instruções necessários para criar a receita de bolo de chocolate. Em seguida, chamamos o método printIngredients no objeto chocolateCake para imprimir os ingredientes na saída do console.

É importante observar que, ao usar funções construtoras para criar objetos, cada instância do objeto criado terá seu próprio conjunto de métodos. Isso pode ser um problema em termos de consumo de memória, especialmente se tivermos muitas instâncias do mesmo objeto. Por isso, a abordagem de criar métodos nos protótipos das funções construtoras pode ser uma maneira mais eficiente de gerenciar o uso de memória.

Aqui está mais um exemplo usando o mesmo tema:

*function* Recipe(*name*, *ingredients*, *instructions*) {

    this.name = *name*;

    this.ingredients = *ingredients*;

    this.instructions = *instructions*;

}

Recipe.prototype.printIngredients = *function* () {

    console.log(`Ingredients for ${this.name}:`);

    for (*let* i = 0; i < this.ingredients.length; i++) {

        console.log(`${i + 1}. ${this.ingredients[i]}`);

    }

};

Recipe.prototype.printInstructions = *function* () {

    console.log(`Instructions for ${this.name}:`);

    console.log(this.instructions);

};

*function* Cake(*name*, *ingredients*, *instructions*, *cakeType*) {

    Recipe.call(this, *name*, *ingredients*, *instructions*);

    this.cakeType = *cakeType*;

}

Cake.prototype = *Object*.create(Recipe.prototype);

Cake.prototype.constructor = Cake;

Cake.prototype.printCakeType = *function* () {

    console.log(`Type of cake: ${this.cakeType}`);

};

*const* chocolateCake = new Cake(

    "Chocolate Cake",

    ["flour", "sugar", "cocoa powder", "baking powder", "eggs", "milk", "vegetable oil"],

    "1. Preheat oven to 350°F. 2. Combine all ingredients in a mixing bowl. 3. Beat with an electric mixer until smooth. 4. Pour into a greased baking dish. 5. Bake for 35-40 minutes or until a toothpick inserted in the center comes out clean.",

    "Layer cake"

);

chocolateCake.printIngredients(); // Output: "Ingredients for Chocolate Cake: 1. flour 2. sugar 3. cocoa powder 4. baking powder 5. eggs 6. milk 7. vegetable oil"

chocolateCake.printInstructions(); // Output: "Instructions for Chocolate Cake: 1. Preheat oven to 350°F. 2. Combine all ingredients in a mixing bowl. 3. Beat with an electric mixer until smooth. 4. Pour into a greased baking dish. 5. Bake for 35-40 minutes or until a toothpick inserted in the center comes out clean."

chocolateCake.printCakeType(); // Output: "Type of cake: Layer cake"

Nesse exemplo, temos a função construtora Recipe, que define uma receita com um nome, ingredientes e instruções. Em seguida, definimos dois métodos no protótipo da função Recipe para imprimir os ingredientes e instruções da receita.

Depois disso, criamos a função construtora Cake, que herda da função construtora Recipe usando o método call para passar os parâmetros da receita para a função Recipe. Em seguida, definimos um método printCakeType no protótipo da função Cake para imprimir o tipo de bolo.

Por fim, criamos um objeto chocolateCake usando a função construtora Cake e passando o nome, ingredientes, instruções e tipo de bolo necessários para criar a receita de bolo de chocolate. Em seguida, chamamos os métodos printIngredients, printInstructions e printCakeType no objeto chocolateCake para imprimir os ingredientes, instruções e tipo de bolo na saída do console.

## Trabalhando com getters e setters em objetos

Getters e setters são métodos especiais em objetos que permitem acessar e modificar propriedades de um objeto de forma controlada. Eles são úteis para garantir que certas propriedades não sejam acessadas ou modificadas indevidamente.

Para criar um getter em um objeto em JavaScript, basta definir uma função que tenha o mesmo nome da propriedade que se deseja acessar e utilizar a palavra-chave get. Já para criar um setter, define-se uma função com o mesmo nome da propriedade e utiliza-se a palavra-chave set.

Veja um exemplo:

*const* pessoa = {

    nome: "João",

    sobrenome: "Silva",

*get* nomeCompleto() {

        return `${this.nome} ${this.sobrenome}`;

    },

*set* nomeCompleto(*valor*) {

*const* partes = *valor*.split(" ");

        this.nome = partes[0];

        this.sobrenome = partes[1];

    }

};

console.log(pessoa.nomeCompleto); // "João Silva"

pessoa.nomeCompleto = "Maria Souza";

console.log(pessoa.nome); // "Maria"

console.log(pessoa.sobrenome); // "Souza"

console.log(pessoa.nomeCompleto); // "Maria Souza"

Nesse exemplo, a propriedade nomeCompleto é uma combinação das propriedades nome e sobrenome. O getter get nomeCompleto() retorna o valor dessa propriedade, enquanto o setter set nomeCompleto(valor) permite modificar o valor da propriedade.

Ao acessar pessoa.nomeCompleto, o getter é chamado e retorna o valor "João Silva". Ao atribuir "Maria Souza" a pessoa.nomeCompleto, o setter é chamado e os valores de nome e sobrenome são modificados.

Isso é útil para garantir que as propriedades sejam modificadas apenas da forma desejada e para garantir que valores inválidos não sejam atribuídos às propriedades.

Outro exemplo usando o tema Biblioteca:

*const* livro = {

    titulo: "A Metamorfose",

    autor: "Franz Kafka",

    \_paginas: 100,

*get* paginas() {

        return this.\_paginas;

    },

*set* paginas(*valor*) {

        if (typeof *valor* !== "number") {

            console.error("O valor precisa ser um número");

            return;

        }

        this.\_paginas = *valor*;

    }

};

console.log(livro.paginas); // 100

livro.paginas = 120;

console.log(livro.paginas); // 120

livro.paginas = "abc"; // "O valor precisa ser um número"

Nesse exemplo, a propriedade \_paginas é privada e só pode ser acessada e modificada através dos getters e setters. O getter get paginas() retorna o valor de \_paginas, enquanto o setter set paginas(valor) permite modificar o valor de \_paginas desde que o novo valor seja um número. Caso contrário, é exibida uma mensagem de erro no console.

Como outro exemplo, podemos criar um objeto que representa um usuário de um fórum na deep web, com as propriedades username, password e email. Podemos utilizar getters e setters para garantir que a senha seja armazenada de forma segura, em formato de hash, e que o email seja mantido em privacidade.

*function* DeepWebUser(*username*, *password*, *email*) {

*let* hashedPassword;

    this.getUsername = *function* () {

        return *username*;

    }

    this.getEmail = *function* () {

        return "Private";

    }

    this.setPassword = *function* (*newPassword*) {

        // Aplicando hash à nova senha antes de armazenar

        hashedPassword = hashFunction(*newPassword*);

    }

    // Função privada para hash da senha

*function* hashFunction(*password*) {

        // Implementação de exemplo, não recomendada para uso real

*let* hash = 0;

        for (*let* i = 0; i < *password*.length; i++) {

            hash += *password*.charCodeAt(i);

        }

        return hash;

    }

}

Nesse exemplo, a propriedade username pode ser acessada através do getter getUsername, a propriedade email é mantida privada através do getter getEmail, e a propriedade password é armazenada de forma segura através do setter setPassword. A implementação do hash usado na função hashFunction é apenas um exemplo, e não deve ser usada em aplicações reais. É importante notar que, em um ambiente real, é necessário aplicar técnicas de criptografia mais seguras e complexas para proteger a senha do usuário.

## Entendendo a diferença entre objetos literais e funções construtoras

Em JavaScript, podemos criar objetos de duas formas diferentes: usando objetos literais ou usando funções construtoras.

Os objetos literais são criados simplesmente definindo um objeto diretamente em uma variável, com suas propriedades e métodos declarados dentro de chaves. Exemplo:

*const* pessoa = {

    nome: "Maria",

    idade: 30,

    saudacao: *function* () {

        console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos.`);

    }

};

Por outro lado, as funções construtoras são criadas através de uma função que, quando chamada com o operador "new", retorna um novo objeto. As propriedades e métodos do objeto são definidos dentro da função, utilizando a palavra-chave this. Exemplo:

*function* Pessoa(*nome*, *idade*) {

    this.nome = *nome*;

    this.idade = *idade*;

    this.saudacao = *function* () {

        console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos.`);

    }

}

*const* pessoa = new Pessoa("Maria", 30);

Uma das principais diferenças entre as duas abordagens é que objetos literais são mais simples e adequados para objetos simples, com poucas propriedades e métodos. Já as funções construtoras são mais adequadas para objetos complexos, com muitas propriedades e métodos, onde é necessário ter um maior controle sobre a criação e inicialização do objeto.

Além disso, as funções construtoras também permitem a criação de herança prototípica, o que pode ser útil em objetos mais complexos. No entanto, as funções construtoras também podem ser mais verbosas e exigir uma sintaxe mais complexa.

Exemplos de objetos literais e funções construtoras com o tema de hardware e software.

Objeto literal:

*const* computador = {

    marca: "Dell",

    processador: "Intel Core i5",

    memoriaRAM: "8GB",

    armazenamento: "256GB SSD",

    ligado: false,

    ligar: *function* () {

        this.ligado = true;

        console.log("O computador foi ligado.");

    },

    desligar: *function* () {

        this.ligado = false;

        console.log("O computador foi desligado.");

    }

};

Função construtora:

*function* Computador(*marca*, *processador*, *memoriaRAM*, *armazenamento*) {

    this.marca = *marca*;

    this.processador = *processador*;

    this.memoriaRAM = *memoriaRAM*;

    this.armazenamento = *armazenamento*;

    this.ligado = false;

    this.ligar = *function* () {

        this.ligado = true;

        console.log("O computador foi ligado.");

    };

    this.desligar = *function* () {

        this.ligado = false;

        console.log("O computador foi desligado.");

    };

}

*const* meuComputador = new Computador("Asus", "Intel Core i7", "16GB", "512GB SSD");

Note que a função construtora permite criar vários objetos do tipo Computador, cada um com suas próprias propriedades e métodos. Isso pode ser útil em um ambiente de desenvolvimento de software onde muitos computadores precisam ser criados e gerenciados de forma programática.

## Implementando interfaces em JavaScript

Em JavaScript, não há um conceito formal de interfaces, como há em outras linguagens de programação, mas podemos implementar algo semelhante usando classes e herança prototípica.

*class* AcessoDeepWeb {

    entrar() {

        throw new *Error*("Método 'entrar' precisa ser implementado.");

    }

}

*class* SiteTor extends AcessoDeepWeb {

    entrar() {

        console.log("Entrando no site através da rede Tor...");

    }

}

*class* SiteI2P extends AcessoDeepWeb {

    entrar() {

        console.log("Entrando no site através da rede I2P...");

    }

}

*const* site1 = new SiteTor();

*const* site2 = new SiteI2P();

site1.entrar(); // "Entrando no site através da rede Tor..."

site2.entrar(); // "Entrando no site através da rede I2P..."

Nesse exemplo, criamos uma interface chamada AcessoDeepWeb, que define um método entrar(). Em seguida, criamos duas classes que implementam essa interface: SiteTor e SiteI2P. Cada uma dessas classes implementa o método entrar() de maneira diferente, de acordo com a rede usada para acessar o site.

Por fim, criamos duas instâncias dessas classes (site1 e site2) e chamamos o método entrar() em cada uma delas. Como ambas as classes implementam a mesma interface, podemos chamar o mesmo método em ambas as instâncias sem nos preocuparmos com as diferenças internas de implementação.

Combinando os temas de Deep Web e Ethical Hacking, podemos criar exemplos de interfaces que definem métodos relacionados à segurança na rede.

Exemplo 1:

*class* Scan {

    execute() {

        throw new *Error*("Método 'execute' precisa ser implementado.");

    }

}

*class* PortScan extends Scan {

    execute() {

        console.log("Executando Port Scan na rede Tor...");

    }

}

*class* VulnerabilityScan extends Scan {

    execute() {

        console.log("Executando Vulnerability Scan na rede I2P...");

    }

}

*const* torScan = new PortScan();

*const* i2pScan = new VulnerabilityScan();

torScan.execute(); // "Executando Port Scan na rede Tor..."

i2pScan.execute(); // "Executando Vulnerability Scan na rede I2P..."

Nesse exemplo, criamos uma interface chamada Scan, que define um método execute(). Em seguida, criamos duas classes que implementam essa interface: PortScan e VulnerabilityScan. Cada uma dessas classes implementa o método execute() de maneira diferente, de acordo com o tipo de scan que está sendo realizado na rede.

Por fim, criamos duas instâncias dessas classes (torScan e i2pScan) e chamamos o método execute() em cada uma delas.

Exemplo 2:

*class* Verificacao {

    checar() {

        throw new *Error*("Método 'checar' precisa ser implementado.");

    }

}

*class* VerificacaoSSL extends Verificacao {

    checar() {

        console.log("Checando SSL na rede Tor...");

    }

}

*class* VerificacaoIP extends Verificacao {

    checar() {

        console.log("Checando IP na rede I2P...");

    }

}

*const* torSSL = new VerificacaoSSL();

*const* i2pIP = new VerificacaoIP();

torSSL.checar(); // "Checando SSL na rede Tor..."

i2pIP.checar(); // "Checando IP na rede I2P..."

Nesse exemplo, criamos uma interface chamada Verificacao, que define um método checar(). Em seguida, criamos duas classes que implementam essa interface: VerificacaoSSL e VerificacaoIP. Cada uma dessas classes implementa o método checar() de maneira diferente, de acordo com o tipo de verificação que está sendo realizada na rede.

Por fim, criamos duas instâncias dessas classes (torSSL e i2pIP) e chamamos o método checar() em cada uma delas.

Exemplo 3:

*class* Ataque {

    realizar() {

        throw new *Error*("Método 'realizar' precisa ser implementado.");

    }

}

*class* AtaqueDDoS extends Ataque {

    realizar() {

        console.log("Realizando ataque DDoS na rede Tor...");

    }

}

*class* AtaqueSQLInjection extends Ataque {

    realizar() {

        console.log("Realizando ataque SQL Injection na rede I2P...");

    }

}

*const* torDDoS = new AtaqueDDoS();

*const* i2pSQLInjection = new AtaqueSQLInjection();

torDDoS.realizar(); // "Realizando ataque DDoS na rede Tor..."

i2pSQLInjection.realizar(); // "Realizando ataque SQL Injection na rede I2P..."

## Usando o método "prototype" para adicionar propriedades e métodos em objetos

O método "prototype" é utilizado em JavaScript para adicionar novas propriedades e métodos a uma classe de objeto. Ele é especialmente útil quando se deseja adicionar novas funcionalidades a objetos existentes sem precisar modificar o objeto original.

Um exemplo de como usar o método "prototype" é o seguinte:

// Definindo a classe de objeto

*function* Carro(*modelo*, *ano*, *cor*) {

    this.modelo = *modelo*;

    this.ano = *ano*;

    this.cor = *cor*;

}

// Adicionando um método à classe de objeto usando prototype

Carro.prototype.descrever = *function* () {

    return "Este carro é um " + this.modelo + " do ano " + this.ano + " e da cor " + this.cor + ".";

};

// Criando um novo objeto da classe Carro

*var* meuCarro = new Carro("Fusca", 1972, "vermelho");

// Chamando o método adicionado usando prototype

console.log(meuCarro.descrever()); // "Este carro é um Fusca do ano 1972 e da cor vermelho."

Neste exemplo, foi definida a classe de objeto "Carro" com as propriedades "modelo", "ano" e "cor". Em seguida, foi adicionado um novo método chamado "descrever" à classe usando o método "prototype". Esse método retorna uma descrição do carro baseado nas suas propriedades.

Depois, um novo objeto chamado "meuCarro" foi criado a partir da classe "Carro" e o método "descrever" foi chamado a partir dele.

O uso do método "prototype" é uma forma poderosa de adicionar funcionalidades a classes de objetos em JavaScript, permitindo uma maior flexibilidade e reutilização de código.

Exemplo usando o tema Reino Monera:

*function* Bacteria(*nome*, *tipo*) {

    this.nome = *nome*;

    this.tipo = *tipo*;

}

Bacteria.prototype.realizarConjugacao = *function* () {

    console.log("A bactéria " + this.nome + " do tipo " + this.tipo + " está realizando a conjugação.");

}

*let* bacteria1 = new Bacteria("E. coli", "Gram-negativa");

*let* bacteria2 = new Bacteria("B. subtilis", "Gram-positiva");

bacteria1.realizarConjugacao(); // "A bactéria E. coli do tipo Gram-negativa está realizando a conjugação."

bacteria2.realizarConjugacao(); // "A bactéria B. subtilis do tipo Gram-positiva está realizando a conjugação."

Nesse exemplo, temos a função construtora Bacteria que recebe dois parâmetros (nome e tipo) e os utiliza para criar um objeto Bacteria. Em seguida, utilizamos o método prototype para adicionar um método realizarConjugacao em todos os objetos Bacteria.

Assim, quando criamos dois objetos Bacteria (bacteria1 e bacteria2) e chamamos o método realizarConjugacao em cada um deles, o resultado é a mensagem indicando que a bactéria está realizando a conjugação.

Vamos criar uma classe Estado que será a classe pai, e duas classes filhas RegiaoNorte e RegiaoNordeste. A classe Estado terá o método getCapital() definido no prototype, e as classes filhas irão estender essa classe e definir o método getPraias() e getClima(), respectivamente.

*class* Estado {

*constructor*(*nome*, *capital*) {

        this.nome = *nome*;

        this.capital = *capital*;

    }

    getCapital() {

        return this.capital;

    }

}

*function* RegiaoNorte(*nome*, *capital*) {

    Estado.call(this, *nome*, *capital*);

}

RegiaoNorte.prototype = *Object*.create(Estado.prototype);

RegiaoNorte.prototype.constructor = RegiaoNorte;

RegiaoNorte.prototype.getPraias = *function* () {

    return "Não há praias na Região Norte.";

}

*function* RegiaoNordeste(*nome*, *capital*) {

    Estado.call(this, *nome*, *capital*);

}

RegiaoNordeste.prototype = *Object*.create(Estado.prototype);

RegiaoNordeste.prototype.constructor = RegiaoNordeste;

RegiaoNordeste.prototype.getClima = *function* () {

    return "O clima da Região Nordeste é predominantemente quente e seco.";

}

*const* amazonas = new RegiaoNorte("Amazonas", "Manaus");

console.log(amazonas.getCapital()); // "Manaus"

console.log(amazonas.getPraias()); // "Não há praias na Região Norte."

*const* pernambuco = new RegiaoNordeste("Pernambuco", "Recife");

console.log(pernambuco.getCapital()); // "Recife"

console.log(pernambuco.getClima()); // "O clima da Região Nordeste é predominantemente quente e seco."

Nesse exemplo, a classe Estado é a classe pai, e as classes RegiaoNorte e RegiaoNordeste são as classes filhas. A classe Estado tem o método getCapital() definido no seu prototype. As classes filhas estendem a classe Estado e definem seus próprios métodos, getPraias() e getClima(), respectivamente. Ao criar instâncias dessas classes, podemos acessar seus métodos específicos.

## Utilizando o padrão de módulo em POO

Suponha que estamos criando um sistema de segurança para uma empresa de tecnologia e precisamos implementar um módulo para gerar senhas seguras. Vamos utilizar o padrão de módulo para encapsular a lógica da geração de senhas em uma classe e torná-la disponível para uso em outras partes do sistema.

Aqui está um exemplo de como isso poderia ser implementado:

*let* PasswordGenerator = (*function* () {

    // constantes privadas

*const* LOWERCASE\_LETTERS = 'abcdefghijklmnopqrstuvwxyz';

*const* UPPERCASE\_LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';

*const* NUMBERS = '0123456789';

*const* SYMBOLS = '!@#$%^&\*()\_+-=[]{}|;:,.<>?';

    // método privado para gerar caracteres aleatórios

*function* getRandomChar(*str*) {

        return *str*.charAt(Math.floor(Math.random() \* *str*.length));

    }

    // classe pública para gerar senhas

*class* PasswordGenerator {

*constructor*(*length* = 8, *options* = {}) {

            this.length = *length*;

            this.includeLowercase = *options*.includeLowercase || true;

            this.includeUppercase = *options*.includeUppercase || true;

            this.includeNumbers = *options*.includeNumbers || true;

            this.includeSymbols = *options*.includeSymbols || false;

        }

        generate() {

*let* charSet = '';

*let* password = '';

            if (this.includeLowercase) {

                charSet += LOWERCASE\_LETTERS;

                password += getRandomChar(LOWERCASE\_LETTERS);

            }

            if (this.includeUppercase) {

                charSet += UPPERCASE\_LETTERS;

                password += getRandomChar(UPPERCASE\_LETTERS);

            }

            if (this.includeNumbers) {

                charSet += NUMBERS;

                password += getRandomChar(NUMBERS);

            }

            if (this.includeSymbols) {

                charSet += SYMBOLS;

                password += getRandomChar(SYMBOLS);

            }

            for (*let* i = password.length; i < this.length; i++) {

                password += getRandomChar(charSet);

            }

            return password;

        }

    }

    // retornar a classe pública

    return PasswordGenerator;

})();

// exemplos de uso

*let* pg1 = new PasswordGenerator(10, { includeSymbols: true });

*let* pg2 = new PasswordGenerator(12, { includeLowercase: false });

console.log(pg1.generate()); // "o%P5!0suqK"

console.log(pg2.generate()); // "V9R&LD23@yA4"

Neste exemplo, a classe PasswordGenerator é definida dentro de uma função anônima que é invocada imediatamente para retornar a classe pública. Isso garante que a classe seja encapsulada e que somente a interface pública seja acessível fora do escopo da função.

A classe PasswordGenerator tem um construtor que aceita um comprimento para a senha e um objeto de opções para indicar quais tipos de caracteres devem ser incluídos na senha. O método generate() é responsável por gerar uma senha aleatória de acordo com as opções fornecidas. A lógica para gerar os caracteres aleatórios é encapsulada em métodos privados.

Ao criar uma instância da classe PasswordGenerator, podemos personalizar as opções de geração de senha, como incluir ou não caracteres maiúsculos, minúsculos e outros.

Vamos supor, agora, que queremos criar um sistema para gerenciar peças de moda íntima feminina. Podemos utilizar o padrão de módulo em POO para criar uma classe que encapsula as funcionalidades de gerenciamento de peças.

Aqui está um exemplo de como podemos fazer isso:

// Módulo para gerenciamento de peças de moda íntima feminina

*const* ModaIntima = (*function* () {

    // Lista de peças de roupa

*let* pecas = [];

    // Classe Peca

*class* Peca {

*constructor*(*nome*, *tamanho*, *cor*, *preco*) {

            this.nome = *nome*;

            this.tamanho = *tamanho*;

            this.cor = *cor*;

            this.preco = *preco*;

        }

        // Retorna uma string com as informações da peça

        toString() {

            return `${this.nome} (${this.tamanho}, ${this.cor}) - R$${this.preco}`;

        }

    }

    // Adiciona uma nova peça à lista

*function* adicionarPeca(*nome*, *tamanho*, *cor*, *preco*) {

*let* peca = new Peca(*nome*, *tamanho*, *cor*, *preco*);

        pecas.push(peca);

    }

    // Remove uma peça da lista

*function* removerPeca(*nome*) {

        pecas = pecas.filter(*peca* *=>* *peca*.nome !== *nome*);

    }

    // Retorna uma lista com todas as peças

*function* listarPecas() {

        return pecas;

    }

    // Retorna uma lista com todas as peças de determinada cor

*function* listarPecasPorCor(*cor*) {

        return pecas.filter(*peca* *=>* *peca*.cor === *cor*);

    }

    // Retorna uma lista com todas as peças de determinado tamanho

*function* listarPecasPorTamanho(*tamanho*) {

        return pecas.filter(*peca* *=>* *peca*.tamanho === *tamanho*);

    }

    // Retorna uma lista com todas as peças ordenadas por preço

*function* listarPecasPorPreco() {

        return pecas.sort((*peca1*, *peca2*) *=>* *peca1*.preco - *peca2*.preco);

    }

    // Retorna o valor total das peças

*function* calcularTotal() {

        return pecas.reduce((*total*, *peca*) *=>* *total* + *peca*.preco, 0);

    }

    // Retorna o número de peças

*function* contarPecas() {

        return pecas.length;

    }

    // Interface pública do módulo

    return {

        adicionarPeca,

        removerPeca,

        listarPecas,

        listarPecasPorCor,

        listarPecasPorTamanho,

        listarPecasPorPreco,

        calcularTotal,

        contarPecas

    };

})();

// Testando o módulo

ModaIntima.adicionarPeca("Calcinha", "M", "Preto", 20.0);

ModaIntima.adicionarPeca("Sutiã", "G", "Branco", 30.0);

ModaIntima.adicionarPeca("Camisola", "P", "Vermelho", 50.0);

console.log(ModaIntima.listarPecas());

// [Peca {nome: "Calcinha", tamanho: "M", cor: "Preto", preco: 20},

//  Peca {nome: "Sutiã", tamanho: "G", cor: "Branco", preco: 30},

//  Peca {nome: "Camisola", cor: "rosa", material: "renda", preco: 89.99, ... }

// Instanciando uma peça

*const* camisola = new Peca("Camisola", "rosa", "renda", 89.99, "P");

// Acessando propriedades da peça

console.log(camisola.nome); // "Camisola"

console.log(camisola.preco); // 89.99

// Usando um método da peça

camisola.calcularDesconto(10); // 80.991

// Criando uma subclasse

*class* Calcinha extends *Peca* {

*constructor*(*nome*, *cor*, *material*, *preco*, *tamanho*, *modelo*) {

        super(*nome*, *cor*, *material*, *preco*, *tamanho*);

        this.modelo = *modelo*;

    }

    mostrarModelo() {

        console.log(O modelo da calcinha é ${ this.modelo }.);

    }

}

// Instanciando uma calcinha

*const* calcinha = new Calcinha("Calcinha fio dental", "preta", "algodão", 15.99, "M", "fio dental");

// Acessando propriedades da calcinha

console.log(calcinha.modelo); // "fio dental"

console.log(calcinha.preco); // 15.99

// Usando um método da calcinha

calcinha.calcularDesconto(5); // 15.19

calcinha.mostrarModelo(); // "O modelo da calcinha é fio dental."

Segue um exemplo com o tema animais silvestres:

// Módulo Animal

*const* Animal = (*function* () {

*const* Animal = *function* (*nome*, *especie*, *habitat*) {

        this.nome = *nome*;

        this.especie = *especie*;

        this.habitat = *habitat*;

    }

    Animal.prototype.comer = *function* () {

        console.log(`${this.nome} está comendo...`);

    }

    Animal.prototype.andar = *function* () {

        console.log(`${this.nome} está andando...`);

    }

    return Animal;

})();

// Módulo Felino

*const* Felino = (*function* () {

*const* Felino = *function* (*nome*, *especie*, *habitat*, *tamanhoGarras*) {

        Animal.call(this, *nome*, *especie*, *habitat*);

        this.tamanhoGarras = *tamanhoGarras*;

    }

    Felino.prototype = *Object*.create(Animal.prototype);

    Felino.prototype.constructor = Felino;

    Felino.prototype.arranhar = *function* () {

        console.log(`${this.nome} está arranhando...`);

    }

    return Felino;

})();

// Instanciando objetos

*const* tigre = new Felino("Tigre", "Panthera tigris", "Florestas tropicais", "15cm");

*const* onca = new Felino("Onça-pintada", "Panthera onca", "Florestas tropicais e savanas", "10cm");

tigre.andar(); // "Tigre está andando..."

onca.comer(); // "Onça-pintada está comendo..."

tigre.arranhar(); // "Tigre está arranhando..."

console.log(tigre.tamanhoGarras); // "15cm"

console.log(onca.habitat); // "Florestas tropicais e savanas"

Neste exemplo, temos dois módulos: Animal e Felino. O módulo Felino herda as propriedades e métodos do módulo Animal e acrescenta a propriedade tamanhoGarras e o método arranhar. Em seguida, instanciamos objetos da classe Felino, que possuem acesso aos métodos e propriedades do módulo Animal e do módulo Felino.

# Manipulação de eventos e call-backs

A manipulação de eventos e callbacks são conceitos importantes em JavaScript, que permitem que o código seja executado de forma assíncrona. Quando um evento ocorre em um elemento HTML, como um clique em um botão, o JavaScript pode ser usado para executar uma função de callback em resposta ao evento.

Um exemplo de manipulação de evento pode ser a criação de um botão que, quando clicado, exibe um alerta com uma mensagem. Para fazer isso, é necessário adicionar um ouvinte de evento ao botão:

//HTML

<button id="meu-botao">Clique aqui</button>

//Javascript

*const* botao = document.querySelector('#meu-botao');

botao.addEventListener('click', *function* () {

    alert('Você clicou no botão!');

});

Nesse exemplo, o método addEventListener é usado para adicionar uma função de callback que será executada quando o botão for clicado. A função de callback simplesmente exibe um alerta com uma mensagem.

Outro exemplo de manipulação de eventos pode ser a adição de um ouvinte de evento para o evento load, que é disparado quando uma página é carregada:

window.addEventListener('load', *function* () {

    console.log('A página foi carregada!');

});

Nesse exemplo, uma função de callback é adicionada como ouvinte de evento para o evento load da janela. Quando a página é carregada, a função de callback é executada e exibe uma mensagem no console.

A manipulação de eventos e callbacks é amplamente utilizada em JavaScript para criar interações dinâmicas em páginas da web e em aplicativos.

## Assíncrona e síncrona

Para exemplificar a diferença entre forma assíncrona e síncrona em JavaScript, vamos considerar o seguinte cenário.

Suponha que temos um sistema que precisa consultar a informação de peixes em um banco de dados. Vamos usar uma API fictícia que simula essa consulta:

*function* consultarPeixes() {

    return new *Promise*((resolve, reject) *=>* {

        setTimeout(() *=>* {

*const* peixes = [

                { nome: "Tilápia", peso: 500 },

                { nome: "Salmão", peso: 1500 },

                { nome: "Pirarucu", peso: 3000 },

            ];

            resolve(peixes);

        }, 2000);

    });

}

Nesse exemplo, a função consultarPeixes() retorna uma Promise que será resolvida após 2 segundos, retornando uma lista de peixes com seus respectivos nomes e pesos.

Vamos supor que precisamos criar uma função que exiba no console o nome e o peso de cada peixe consultado. Podemos fazer isso de duas formas: de forma assíncrona e síncrona.

### Forma assíncrona:

*function* exibirPeixesAssincronamente() {

    console.log("Consultando peixes...");

    consultarPeixes()

        .then(*peixes* *=>* {

            console.log("Peixes consultados:");

*peixes*.forEach(*peixe* *=>* {

                console.log(`- ${*peixe*.nome}: ${*peixe*.peso}g`);

            });

        })

        .catch(*error* *=>* {

            console.error(*error*);

        });

}

Nesse exemplo, a função exibirPeixesAssincronamente() exibe a mensagem "Consultando peixes..." no console, e em seguida chama a função consultarPeixes() que retorna uma Promise. A função .then() é usada para tratar a resolução da Promise, e exibe no console o nome e o peso de cada peixe consultado.

### Forma síncrona:

async *function* exibirPeixesSincronamente() {

    console.log("Consultando peixes...");

    try {

*const* peixes = await consultarPeixes();

        console.log("Peixes consultados:");

        peixes.forEach(*peixe* *=>* {

            console.log(`- ${*peixe*.nome}: ${*peixe*.peso}g`);

        });

    } catch (error) {

        console.error(error);

    }

}

Nesse exemplo, a função exibirPeixesSincronamente() é marcada como async, permitindo o uso do operador await. Dessa forma, podemos fazer uma chamada assíncrona da função consultarPeixes() e aguardar a sua resolução antes de continuar o código. A função try...catch é usada para tratar eventuais erros na consulta. Em seguida, a função exibe no console o nome e o peso de cada peixe consultado.

Perceba que na forma assíncrona, a consulta é realizada em paralelo com a execução do restante do código, enquanto na forma síncrona, o código aguarda a resolução da Promise antes de continuar a execução.

Em geral, a forma assíncrona é mais indicada em casos em que não queremos bloquear a execução do código enquanto uma operação é realizada, enquanto a forma síncrona é mais indicada em casos em que precisamos esperar a conclusão de uma operação antes de continuar com o restante do código.

No JavaScript, uma das principais maneiras de implementar a forma assíncrona é através do uso de callbacks. Um callback é uma função que é passada como argumento para outra função e que é executada quando um determinado evento ocorre.

Por exemplo, em operações de requisição de dados em APIs, podemos usar callbacks para executar uma ação quando a resposta da requisição estiver disponível. Veja o exemplo abaixo:

*function* requisitarDados(*url*, *callback*) {

    fetch(*url*)

        .then(*response* *=>* *response*.json())

        .then(*data* *=>* callback(*data*))

        .catch(*error* *=>* console.log(*error*))

}

*function* tratarDados(*data*) {

    console.log(*data*)

}

requisitarDados('https://api.minhaapi.com.br/dados', tratarDados);

Nesse exemplo, a função requisitarDados é responsável por fazer uma requisição à API e, quando a resposta estiver disponível, chamar o callback tratarDados passando os dados da resposta como argumento. A função tratarDados é definida separadamente e pode ser reutilizada em outras partes do código.

Além disso, o JavaScript também oferece outras formas de trabalhar com assincronia, como Promises e async/await, que são recursos mais modernos e que permitem uma escrita mais clara e fácil de entender.

## Manipulação de eventos em um formulário

Vamos criar um exemplo de manipulação de eventos em um formulário.

Suponha que temos um formulário para cadastro de voluntários que querem ajudar a limpar os rios do Brasil. Ao enviar o formulário, queremos validar se os campos obrigatórios foram preenchidos e exibir uma mensagem de sucesso ou erro de acordo com o resultado da validação.

Para isso, vamos utilizar o evento "submit" do formulário e um callback para manipular o evento.

Exemplo de HTML do formulário:

<form id="form-voluntarios">

  <label for="nome">Nome:</label>

  <input type="text" name="nome" id="nome" required>

  <label for="email">Email:</label>

  <input type="email" name="email" id="email" required>

  <label for="cidade">Cidade:</label>

  <input type="text" name="cidade" id="cidade" required>

  <button type="submit">Enviar</button>

</form>

Exemplo de código JavaScript para manipulação do evento:

*const* formVoluntarios = document.querySelector('#form-voluntarios');

formVoluntarios.addEventListener('submit', *function* (*event*) {

*event*.preventDefault(); // previne o envio do formulário

*const* nome = document.querySelector('#nome').value;

*const* email = document.querySelector('#email').value;

*const* cidade = document.querySelector('#cidade').value;

    // Verifica se os campos obrigatórios foram preenchidos

    if (!nome || !email || !cidade) {

        alert('Por favor, preencha todos os campos obrigatórios.');

        return;

    }

    // Envia o formulário

    // Aqui você pode chamar uma API para salvar os dados do voluntário, por exemplo

    alert('Cadastro realizado com sucesso!');

});

Nesse exemplo, utilizamos o método addEventListener para adicionar um callback ao evento "submit" do formulário. Dentro do callback, utilizamos o método event.preventDefault() para prevenir o envio do formulário e fazer a validação dos campos obrigatórios. Caso algum campo esteja vazio, exibimos um alerta com a mensagem de erro. Caso contrário, exibimos um alerta com a mensagem de sucesso e podemos enviar o formulário para uma API, por exemplo.

## Callbacks

Em JavaScript, callbacks são funções que são passadas como argumentos para outras funções e são executadas quando a função original é concluída. Os callbacks são amplamente utilizados em JavaScript para executar operações assíncronas, como fazer solicitações de rede ou ler e gravar arquivos.

Um exemplo de callback em JavaScript seria o uso da função setTimeout(). Essa função é usada para atrasar a execução de uma função por um determinado tempo. O segundo argumento da função setTimeout() é um callback que será executado após o tempo especificado ter decorrido.

Por exemplo:

setTimeout(() *=>* {

    console.log('Este é um exemplo de callback');

}, 3000);

Nesse exemplo, uma função anônima é passada como argumento para setTimeout(). Essa função será executada depois de um atraso de 3 segundos (3000 milissegundos). Quando a função é executada, a mensagem "Este é um exemplo de callback" é exibida no console.

Outro exemplo comum de callback é o uso da função forEach() para iterar sobre uma matriz. A função forEach() recebe uma função de callback como argumento e executa essa função uma vez para cada elemento da matriz. Por exemplo:

*const* animais = ['Leão', 'Tigre', 'Elefante'];

animais.forEach((*animal*) *=>* {

    console.log(*animal*);

});

Nesse exemplo, a função forEach() recebe uma função anônima como argumento. Essa função é executada uma vez para cada elemento na matriz animais. A mensagem "Leão", "Tigre" e "Elefante" é exibida no console.

Existem muitos tipos de callbacks em JavaScript, mas alguns dos mais comuns são:

**Callbacks de tempo:** são funções que são executadas após um determinado período de tempo, usando as funções setTimeout e setInterval.

console.log("Iniciando contagem regressiva...");

setTimeout(*function* () {

    console.log("10 segundos restantes...");

}, 5000);

Neste exemplo, a mensagem "10 segundos restantes..." será exibida após 5 segundos.

**Callbacks de eventos**: são funções que são executadas quando um evento ocorre, como um clique de mouse ou um carregamento de página. Eles são usados com o método addEventListener.

document.addEventListener("keydown", *function* (*event*) {

    console.log("Tecla pressionada: " + *event*.keyCode);

});

Neste exemplo, a função é executada sempre que uma tecla é pressionada.

**Callbacks de retorno de chamada**: são funções que são passadas como argumentos para outras funções e executadas quando a função principal é concluída.

*function* buscarDados(*url*, *sucesso*, *erro*) {

*let* xhr = new *XMLHttpRequest*();

    xhr.open("GET", *url*);

    xhr.onload = *function* () {

        if (xhr.status === 200) {

            sucesso(xhr.response);

        } else {

            erro(xhr.status);

        }

    };

    xhr.send();

}

buscarDados("http://meusite.com/dados", *function* (*resultado*) {

    console.log("Dados recebidos: " + *resultado*);

}, *function* (*erro*) {

    console.log("Erro ao buscar dados. Código: " + *erro*);

});

Neste exemplo, a função buscarDados recebe dois callbacks como argumentos: sucesso e erro, que são executados dependendo do resultado da solicitação.

# Trabalhando com Promises e Async/Await

As Promises e o Async/Await são recursos do JavaScript para lidar com operações assíncronas de maneira mais simples e legível.

As Promises permitem que uma operação assíncrona possa retornar um valor ou um erro quando a operação for concluída, de forma que é possível definir o que deve acontecer após o término da operação.

Por exemplo:

*function* fazerRequisicao(*url*) {

    return new *Promise*((resolve, reject) *=>* {

*const* xhr = new *XMLHttpRequest*();

        xhr.open('GET', *url*);

        xhr.onload = () *=>* resolve(xhr.responseText);

        xhr.onerror = () *=>* reject(xhr.statusText);

        xhr.send();

    });

}

fazerRequisicao('https://api.github.com/users')

    .then((*response*) *=>* console.log(*response*))

    .catch((*error*) *=>* console.log(*error*));

Nesse exemplo, a função fazerRequisicao recebe uma URL e retorna uma Promise que faz uma requisição AJAX e resolve com a resposta ou rejeita com o status de erro, dependendo do resultado da requisição.

O método then é utilizado para lidar com o resultado da Promise quando ela é resolvida, e o método catch é utilizado para lidar com o resultado quando ela é rejeitada.

Já o Async/Await é uma sintaxe mais recente do JavaScript que permite escrever código assíncrono de forma mais parecida com código síncrono, utilizando a palavra-chave async para definir uma função como assíncrona e a palavra-chave await para aguardar o resultado de uma Promise antes de prosseguir com a execução do código. Por exemplo:

async *function* fazerRequisicao(*url*) {

*const* response = await fetch(*url*);

*const* data = await response.json();

    return data;

}

fazerRequisicao('https://api.github.com/users')

    .then((*data*) *=>* console.log(*data*))

    .catch((*error*) *=>* console.log(*error*));

Nesse exemplo, a função fazerRequisicao é definida como assíncrona e utiliza a palavra-chave await para aguardar o resultado da requisição AJAX antes de prosseguir com a execução do código. O método then é utilizado para lidar com o resultado da função quando ela é resolvida, e o método catch é utilizado para lidar com o resultado quando ela é rejeitada.

Mais alguns exemplos:

**Exemplo usando a API da NASA:**

async *function* fetchAPOD() {

    try {

*const* response = await fetch('https://api.nasa.gov/planetary/apod?api\_key=DEMO\_KEY');

*const* data = await response.json();

        console.log(data);

    } catch (error) {

        console.error(error);

    }

}

fetchAPOD();

Nesse exemplo, estamos usando a API da NASA para buscar a Astronomy Picture of the Day (APOD). Estamos utilizando async/await para fazer a requisição de forma assíncrona e tratando os erros com try/catch.

**Exemplo usando a API da Wikipédia:**

async *function* fetchArticle(*title*) {

    try {

*const* response = await fetch(`https://en.wikipedia.org/api/rest\_v1/page/html/${*title*}`);

*const* html = await response.text();

        console.log(html);

    } catch (error) {

        console.error(error);

    }

}

fetchArticle('JavaScript');

Nesse exemplo, estamos usando a API da Wikipédia para buscar o artigo sobre JavaScript. Novamente, estamos utilizando async/await para fazer a requisição de forma assíncrona e tratando os erros com try/catch. Nesse caso, estamos buscando o HTML do artigo ao invés de um JSON como na API da NASA.

Veja um exemplo de projeto completo usando a API de Pokemons mostrando-os numa galeria com HTML5 e CSS3.

Antes de começar, é importante destacar que será necessário conhecimentos em HTML, CSS e JavaScript para seguir este projeto. Além disso, para consumir a API de Pokemons, será utilizado o método fetch() do JavaScript para buscar dados na API e apresentá-los na página.

Passo 1: Configurando a estrutura HTML

Começaremos criando a estrutura básica do HTML, que consistirá em uma seção para a galeria de Pokemons e outra para exibir informações sobre o Pokemon selecionado.

<!DOCTYPE html>

<html>

  <head>

    <title>Lista de Pokémons</title>

    <link rel="stylesheet" type="text/css" href="assets/css/style.css">

  </head>

  <body>

    <h1>Lista de Pokémons</h1>

    <div id="pokemon-list"></div>

    <script src="assets/js/script.js"></script>

  </body>

</html>

Passo 2: Estilizando a página com CSS3

Vamos estilizar a página utilizando CSS3, para dar uma aparência mais agradável e interessante à nossa galeria de Pokemons. Para isso, criaremos um arquivo style.css na raiz do projeto e o importaremos em nossa página HTML, como está no código acima.

Agora, no arquivo style.css, vamos definir algumas regras de estilo para a nossa página:

body {

    margin: 0;

    padding: 0;

    display: flex;

    flex-direction: column;

    align-items: center;

    justify-content: center;

    height: 100vh;

    background-color: #f1f1f1;

    font-family: Arial, sans-serif;

  }

  h1 {

    text-align: center;

    text-shadow: 2px 2px 4px #000000;

  }

  #pokemon-list {

    display: flex;

    flex-wrap: wrap;

    justify-content: center;

  }

  .pokemon {

    margin: 20px;

    padding: 20px;

    border-radius: 10px;

    box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.2);

    transition: all 0.3s ease;

    text-align: center;

    background-color: #ffffff;

    width: 200px;

  }

  .pokemon:hover {

    transform: translateY(-10px);

    box-shadow: 0px 10px 20px rgba(0, 0, 0, 0.4);

  }

  .pokemon img {

    width: 100%;

    height: auto;

  }

  .pokemon h2 {

    margin-top: 10px;

    margin-bottom: 5px;

  }

  .pokemon p {

    margin: 5px;

  }

Passo 3: O arquivo javascript

*const* pokemonList = document.getElementById("pokemon-list");

fetch("https://pokeapi.co/api/v2/pokemon?limit=1118")

    .then(*response* *=>* *response*.json())

    .then(*data* *=>* {

*const* pokemons = *data*.results;

        pokemons.forEach(*pokemon* *=>* {

            fetch(*pokemon*.url)

                .then(*response* *=>* *response*.json())

                .then(*data* *=>* {

*const* name = *data*.name;

*const* image = *data*.sprites.front\_default;

*const* abilities = *data*.abilities.map(*ability* *=>* *ability*.ability.name).join(", ");

*const* pokemonElement = document.createElement("div");

                    pokemonElement.classList.add("pokemon");

                    pokemonElement.innerHTML = `

            <h2>${name}</h2>

            <img src="${image}" alt="${name}">

            <p><strong>Abilities:</strong> ${abilities}</p>

          `;

                    pokemonList.appendChild(pokemonElement);

                });

        });

    });

Certifique-se de salvar os arquivos corretamente em suas respectivas pastas e ajuste os links no index.html de acordo com a sua estrutura de pasta.

Outro exemplo de projeto que usa uma API de fontes de notícias para mostrar notícias sobre ciência e tecnologia.

**Passo 1: Criando a estrutura básica do projeto**

Para começar, vamos criar uma pasta para o nosso projeto e dentro dela vamos criar um arquivo HTML, um arquivo CSS e um arquivo JavaScript. O arquivo HTML vai conter a estrutura básica da página, o arquivo CSS vai conter a estilização e o arquivo JavaScript vai conter a lógica da aplicação.

meu-projeto/

├── index.html

├── style.css

└── script.js

**Passo 2: Obtendo a chave da API**

Para usar a API de fontes de notícias, precisamos obter uma chave de acesso. Nesse caso, vamos utilizar a API do NewsAPI (https://newsapi.org/). Para obter a chave, siga os seguintes passos:

1. Acesse o site https://newsapi.org/
2. Clique no botão "Get API Key" no canto superior direito
3. Siga as instruções para criar uma conta e obter a chave da API

Passo 3: Conectando com a API

Com a chave da API em mãos, podemos começar a conectar com a API para obter as notícias de ciência e tecnologia. Para isso, vamos criar uma função assíncrona getNews que recebe como parâmetro o tema das notícias (que pode ser "science" ou "technology") e retorna uma Promise com os dados das notícias.

async *function* getNews(*topic*) {

*const* apiKey = "sua\_chave\_da\_api\_aqui";

*const* url = `https://newsapi.org/v2/top-headlines?country=br&category=${*topic*}&apiKey=${apiKey}`;

*const* response = await fetch(url);

*const* data = await response.json();

    return data.articles;

}

Nessa função, criamos a URL da API com a chave da API e o tema das notícias passado como parâmetro. Depois, fazemos uma requisição usando o fetch e esperamos a resposta usando o await. Por fim, retornamos apenas o array de notícias.

Passo 4: Renderizando as notícias na página

Com os dados das notícias em mãos, podemos renderizá-los na página HTML. Para isso, vamos criar uma função renderNews que recebe como parâmetro um array de notícias e atualiza a lista de notícias na página HTML.

*function* renderNews(*news*) {

*const* newsList = document.querySelector(".news-list");

    newsList.innerHTML = "";

*news*.forEach((*article*) *=>* {

*const* li = document.createElement("li");

*const* title = document.createElement("h2");

*const* summary = document.createElement("p");

*const* link = document.createElement("a");

        title.textContent = *article*.title;

        summary.textContent = *article*.description;

        link.textContent = "Leia mais";

        link.href = *article*.url;

        link.target = "\_blank";

        li.appendChild(title);

        li.appendChild(summary);

        li.appendChild(link);

        newsList.appendChild(li);

    });

}

Nessa função, limpamos a lista de notícias na página HTML e, para cada notícia recebida como parâmetro, criamos uma nova li com um h2 para o título, um p para o resumo e um link para a notícia completa.

Segue abaixo um exemplo de como poderia ser feito o CSS para estilizar o projeto com a API de fontes de notícias acima:

/\* Estilos gerais \*/

body {

    font-family: Arial, sans-serif;

    margin: 0;

    padding: 0;

    background-color: #f0f0f0;

  }

  .container {

    max-width: 960px;

    margin: 0 auto;

    padding: 20px;

  }

  h1 {

    text-align: center;

    font-size: 36px;

    margin-bottom: 30px;

  }

  /\* Estilos da lista de notícias \*/

  ul {

    list-style: none;

    padding: 0;

    margin: 0;

  }

  li {

    background-color: #fff;

    margin-bottom: 20px;

    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);

    border-radius: 10px;

  }

  a {

    text-decoration: none;

    color: #333;

  }

  a:hover {

    text-decoration: underline;

  }

  img {

    width: 100%;

    height: auto;

    border-top-left-radius: 10px;

    border-top-right-radius: 10px;

  }

  h2 {

    font-size: 24px;

    margin: 20px;

  }

  p {

    margin: 0 20px 20px;

  }

  button {

    display: block;

    margin: 20px auto;

    padding: 10px 20px;

    background-color: #333;

    color: #fff;

    border: none;

    border-radius: 20px;

    cursor: pointer;

  }

  button:hover {

    background-color: #444;

  }

  /\* Estilos do carregamento \*/

  .loading {

    text-align: center;

    font-size: 24px;

    margin: 50px 0;

  }

  .loading img {

    width: 100px;

    height: 100px;

    animation: spin 1s infinite linear;

  }

  @keyframes spin {

    from {

      transform: rotate(0deg);

    }

    to {

      transform: rotate(360deg);

    }

  }

Lembrando que estes estilos são apenas um exemplo e podem ser alterados de acordo com a preferência do desenvolvedor.

Existem várias outras APIs de notícias que podem ser utilizadas como alternativas, algumas delas são:

1. News API: https://newsapi.org/
2. Google News API: https://developers.google.com/news
3. Bing News Search API: https://azure.microsoft.com/en-us/services/cognitive-services/bing-news-search-api/
4. NewsCatcher API: https://newscatcherapi.com/
5. ContextualWeb News API: https://contextualweb.io/news-api/
6. Gnews API: https://gnews.io/

Cada API possui suas próprias especificações e documentação, então é importante ler a documentação para entender como utilizá-las corretamente.

# Programação Funcional em JavaScript

A programação funcional é um paradigma de programação que se concentra no uso de funções para criar programas. Em vez de alterar o estado de variáveis ​​como ocorre na programação imperativa, a programação funcional se concentra em escrever funções puras que não possuem efeitos colaterais e que operam apenas nos seus argumentos. JavaScript suporta programação funcional e é possível escrever programas neste estilo usando uma variedade de recursos da linguagem.

Algumas das características da programação funcional que podem ser utilizadas em JavaScript incluem:

**Funções puras:** funções que não têm efeitos colaterais e retornam sempre o mesmo resultado para os mesmos argumentos.

**Imutabilidade:** o uso de variáveis imutáveis, ou seja, que não podem ser alteradas após a sua criação.

**Composição de funções:** a construção de funções mais complexas a partir de funções menores e mais simples.

**Recursão:** a definição de funções que chamam a si mesmas para resolver problemas.

**Closures:** o uso de funções que capturam o estado do ambiente em que foram criadas.

Algumas das funções nativas do JavaScript que podem ser utilizadas em programação funcional incluem **map(), reduce(), filter() e forEach()**. Estas funções permitem realizar operações em listas de forma mais concisa e declarativa.

Alguns frameworks populares para programação funcional em JavaScript incluem React e Redux. Esses frameworks utilizam conceitos como imutabilidade e composição de funções para criar interfaces de usuário mais eficientes e escaláveis.

## Funções puras em JS

Em JavaScript, uma função pura é uma função que não tem efeitos colaterais e sempre retorna o mesmo resultado para os mesmos argumentos. Isso significa que, quando uma função pura é chamada com os mesmos argumentos, ela sempre retorna o mesmo resultado e não altera nenhum estado externo.

Algumas características de uma função pura incluem:

**Não altera nenhum estado externo**: uma função pura não modifica variáveis ​​globais ou qualquer estado externo à função.

**Sem efeitos colaterais:** uma função pura não tem efeitos colaterais, o que significa que ela não produz nenhum efeito além de retornar um valor.

**Sempre retorna o mesmo resultado:** uma função pura sempre retorna o mesmo resultado para os mesmos argumentos.

Aqui está um exemplo de uma função impura e de como ela pode ser transformada em uma função pura:

// Função impura - Modifica variável global

*let* counter = 0;

*function* increment() {

    counter++;

    return counter;

}

// Função pura - Não modifica variável global

*function* incrementPure(*counter*) {

    return counter + 1;

}

No exemplo acima, a função increment é impura porque modifica a variável global counter. Em contraste, a função incrementPure é pura porque não modifica nenhum estado externo e sempre retorna o mesmo resultado para os mesmos argumentos.

As funções puras são importantes em programação funcional porque permitem que o código seja mais fácil de entender e de testar. Como as funções puras não têm efeitos colaterais e sempre retornam o mesmo resultado, é mais fácil prever o comportamento do código e testar cada função individualmente.

Exemplos de funções puras usando o tema universidades:

**Exemplo 1: Função para verificar se um aluno está matriculado em um curso**

*const* cursosMatriculados = ['Matemática', 'Física', 'Química'];

*function* verificarMatricula(*cursosMatriculados*, *curso*) {

    return cursosMatriculados.includes(curso);

}

Neste exemplo, a função verificarMatricula é pura porque ela não modifica nenhum estado externo e sempre retorna o mesmo resultado para os mesmos argumentos. A função simplesmente verifica se um aluno está matriculado em um determinado curso, comparando o nome do curso com a lista de cursos matriculados.

**Exemplo 2: Função para classificar livros por autor**

*const* livros = [

    { titulo: 'O Grande Gatsby', autor: 'F. Scott Fitzgerald' },

    { titulo: 'Dom Casmurro', autor: 'Machado de Assis' },

    { titulo: 'Orgulho e Preconceito', autor: 'Jane Austen' }

];

*function* classificarPorAutor(*livros*) {

    return livros.slice().sort((*a*, *b*) *=>* a.autor.localeCompare(b.autor));

}

Neste exemplo, a função classificarPorAutor é pura porque ela não modifica nenhum estado externo e sempre retorna o mesmo resultado para os mesmos argumentos. A função cria uma cópia da lista de livros, classifica os livros por autor e retorna a lista classificada.

**Exemplo 3: Função para verificar se uma tarefa foi concluída**

*const* tarefasConcluidas = ['Tarefa 1', 'Tarefa 2', 'Tarefa 3'];

*function* verificarConclusaoTarefa(*tarefasConcluidas*, *tarefa*) {

  return tarefasConcluidas.includes(tarefa);

}

Neste exemplo, a função verificarConclusaoTarefa é pura porque ela não modifica nenhum estado externo e sempre retorna o mesmo resultado para os mesmos argumentos. A função simplesmente verifica se uma determinada tarefa foi concluída pelo aluno, comparando o nome da tarefa com a lista de tarefas concluídas.

## Imutabilidade

Aqui estão alguns exemplos de como a imutabilidade é usada na programação funcional em JavaScript, usando o tema de universidades dos EUA:

**Exemplo 1: Imutabilidade em uma lista de universidades**

*const* universidades = Object.freeze([

    { nome: 'Universidade de Harvard', estado: 'Massachusetts' },

    { nome: 'Universidade de Yale', estado: 'Connecticut' },

    { nome: 'Universidade de Stanford', estado: 'Califórnia' }

]);

// Adicionando uma nova universidade à lista

*const* novaUniversidade = { nome: 'Universidade de Princeton', estado: 'Nova Jersey' };

*const* novasUniversidades = [...universidades, novaUniversidade];

Neste exemplo, a lista de universidades é definida como const e envolvida na função Object.freeze() para torná-la imutável. A adição de uma nova universidade é feita criando uma nova lista com a sintaxe de espalhamento (...universidades) e adicionando a nova universidade ao final da lista. Isso cria uma nova lista em vez de modificar a lista original.

**Exemplo 2: Imutabilidade em uma função de filtragem**

*const* universidades = [

    { nome: 'Universidade de Harvard', estado: 'Massachusetts' },

    { nome: 'Universidade de Yale', estado: 'Connecticut' },

    { nome: 'Universidade de Stanford', estado: 'Califórnia' }

];

*function* filtrarPorEstado(*universidades*, *estado*) {

    return Object.freeze(universidades.filter(*u* *=>* u.estado === estado));

}

// Filtrando as universidades por estado

*const* universidadesEmMassachusetts = filtrarPorEstado(universidades, 'Massachusetts');

Neste exemplo, a função filtrarPorEstado recebe uma lista de universidades e um estado como argumentos e retorna uma nova lista filtrada com as universidades que estão no estado especificado. A lista de universidades original não é modificada, já que a função Object.freeze() é usada para tornar a lista imutável antes de fazer a filtragem.

**Exemplo 3: Imutabilidade em uma função de ordenação**

*const* universidades = [

    { nome: 'Universidade de Harvard', estado: 'Massachusetts', classificacao: 1 },

    { nome: 'Universidade de Yale', estado: 'Connecticut', classificacao: 3 },

    { nome: 'Universidade de Stanford', estado: 'Califórnia', classificacao: 2 }

];

*function* ordenarPorClassificacao(*universidades*) {

    return Object.freeze([...universidades].sort((*a*, *b*) *=>* a.classificacao - b.classificacao));

}

// Ordenando as universidades por classificação

*const* universidadesOrdenadas = ordenarPorClassificacao(universidades);

Neste exemplo, a função ordenarPorClassificacao recebe uma lista de universidades e retorna uma nova lista ordenada com as universidades classificadas por sua classificação. A lista de universidades original é protegida com a função Object.freeze() e, em seguida, é feita uma cópia da lista usando a sintaxe de espalhamento ([...universidades]). A função sort() é usada para classificar as universidades por sua classificação e retornar uma nova lista ordenada.

Veja mais dois exemplos de como a imutabilidade pode ser usada na programação funcional em JavaScript, usando o tema de universidades dos EUA:

**Exemplo 4: Imutabilidade em uma função de map**

*const* universidades = [

    { nome: 'Universidade de Harvard', estado: 'Massachusetts', classificacao: 1 },

    { nome: 'Universidade de Yale', estado: 'Connecticut', classificacao: 3 },

    { nome: 'Universidade de Stanford', estado: 'Califórnia', classificacao: 2 }

];

*function* atualizarClassificacao(*universidades*, *nome*, *novaClassificacao*) {

    return Object.freeze(

        universidades.map(*u* *=>* u.nome === nome ? { ...u, classificacao: novaClassificacao } : u)

    );

}

// Atualizando a classificação da Universidade de Yale

*const* universidadesAtualizadas = atualizarClassificacao(universidades, 'Universidade de Yale', 2);

Neste exemplo, a função atualizarClassificacao recebe uma lista de universidades, o nome de uma universidade e uma nova classificação como argumentos. A função retorna uma nova lista de universidades onde a universidade com o nome especificado tem sua classificação atualizada para o novo valor. A lista de universidades original não é modificada, já que a função Object.freeze() é usada para torná-la imutável. A função map() é usada para criar uma nova lista de universidades, onde a universidade com o nome especificado tem sua classificação atualizada e as outras universidades são mantidas intactas.

**Exemplo 5: Imutabilidade em uma função de redução**

*const* universidades = [

    { nome: 'Universidade de Harvard', estado: 'Massachusetts', alunos: 21000 },

    { nome: 'Universidade de Yale', estado: 'Connecticut', alunos: 13000 },

    { nome: 'Universidade de Stanford', estado: 'Califórnia', alunos: 17000 }

];

*function* totalAlunos(*universidades*) {

    return Object.freeze(

        universidades.reduce((*total*, *u*) *=>* total + u.alunos, 0)

    );

}

// Obtendo o total de alunos de todas as universidades

*const* total = totalAlunos(universidades);

Neste exemplo, a função totalAlunos recebe uma lista de universidades como argumento e retorna o número total de alunos em todas as universidades. A lista de universidades original é protegida com a função Object.freeze() para torná-la imutável. A função reduce() é usada para iterar sobre a lista de universidades e acumular o número de alunos em cada uma delas em uma variável total.

A variável total é retornada como o resultado da função. Como a função reduce() não modifica a lista original de universidades e a função Object.freeze() protege a lista de ser modificada, a função totalAlunos é uma função pura.

## Composição de funções

Composição de funções na programação funcional em javascript, cite diversos códigos de exemplo usando o tema países da Europa

Composição de funções é uma técnica muito utilizada na programação funcional em JavaScript para criar funções complexas a partir da combinação de funções menores e mais simples. Aqui estão alguns exemplos de como a composição de funções pode ser usada com o tema de países da Europa:

**Exemplo 1: Composição de funções para obter a capital de um país**

*const* paises = [

    { nome: 'França', capital: 'Paris' },

    { nome: 'Alemanha', capital: 'Berlim' },

    { nome: 'Itália', capital: 'Roma' }

];

*const* obterCapital = *pais* *=>* pais.capital;

*const* encontrarPais = *nome* *=>* paises.find(*pais* *=>* pais.nome === nome);

*const* obterCapitalDoPais = *nome* *=>* obterCapital(encontrarPais(nome));

// Obtendo a capital da França

*const* capitalDaFranca = obterCapitalDoPais('França'); // 'Paris'

Neste exemplo, a função obterCapitalDoPais é criada a partir da composição das funções encontrarPais e obterCapital. A função encontrarPais recebe o nome de um país e retorna o objeto país correspondente da lista paises. A função obterCapital recebe um objeto país e retorna o nome da capital desse país. A função obterCapitalDoPais recebe o nome de um país, encontra o objeto país correspondente usando a função encontrarPais e retorna o nome da capital desse país usando a função obterCapital.

**Exemplo 2: Composição de funções para verificar se um país está na União Europeia**

*const* paises = [

    { nome: 'França', ue: true },

    { nome: 'Alemanha', ue: true },

    { nome: 'Itália', ue: true },

    { nome: 'Reino Unido', ue: false }

];

*const* encontrarPais = *nome* *=>* paises.find(*pais* *=>* pais.nome === nome);

*const* verificarUE = *pais* *=>* pais.ue;

*const* paisEstaNaUE = *nome* *=>* verificarUE(encontrarPais(nome));

// Verificando se a Itália está na União Europeia

*const* italiaNaUE = paisEstaNaUE('Itália'); // true

Neste exemplo, a função paisEstaNaUE é criada a partir da composição das funções encontrarPais e verificarUE. A função encontrarPais recebe o nome de um país e retorna o objeto país correspondente da lista paises. A função verificarUE recebe um objeto país e retorna um valor booleano indicando se o país está na União Europeia ou não. A função paisEstaNaUE recebe o nome de um país, encontra o objeto país correspondente usando a função encontrarPais e verifica se o país está na União Europeia ou não usando a função verificarUE.

Aqui estão mais alguns exemplos de como a composição de funções pode ser usada com o tema Reino Plantae:

**Exemplo 1: Composição de funções para filtrar plantas por família**

*const* plantas = [

    { nome: 'Rosa', familia: 'Rosaceae', altura: 1.5 },

    { nome: 'Girassol', familia: 'Asteraceae', altura: 2.0 },

    { nome: 'Orquídea', familia: 'Orchidaceae', altura: 0.5 }

];

*const* filtrarPorFamilia = *familia* *=>* *planta* *=>* planta.familia === familia;

*const* obterNomes = *planta* *=>* planta.nome;

*const* plantasDaFamiliaRosaceae = plantas.filter(filtrarPorFamilia('Rosaceae')).map(obterNomes);

console.log(plantasDaFamiliaRosaceae); // ['Rosa']

Neste exemplo, a função filtrarPorFamilia recebe o nome de uma família de plantas e retorna uma função que filtra as plantas da lista plantas que pertencem a essa família. A função obterNomes recebe um objeto planta e retorna o nome da planta. A constante plantasDaFamiliaRosaceae é criada a partir da composição das funções filtrarPorFamilia e obterNomes. Ela filtra as plantas da família Rosaceae usando a função filtrarPorFamilia e retorna apenas os nomes dessas plantas usando a função obterNomes.

**Exemplo 2: Composição de funções para obter a média de altura das plantas**

*const* plantas = [

    { nome: 'Rosa', familia: 'Rosaceae', altura: 1.5 },

    { nome: 'Girassol', familia: 'Asteraceae', altura: 2.0 },

    { nome: 'Orquídea', familia: 'Orchidaceae', altura: 0.5 }

];

*const* obterAlturas = *planta* *=>* planta.altura;

*const* somarAlturas = (*acumulador*, *altura*) *=>* acumulador + altura;

*const* calcularMedia = *valores* *=>* valores.reduce(somarAlturas, 0) / valores.length;

*const* mediaDeAlturaDasPlantas = calcularMedia(plantas.map(obterAlturas));

console.log(mediaDeAlturaDasPlantas); // 1.3333333333333333

Neste exemplo, a função obterAlturas recebe um objeto planta e retorna a altura da planta. A função somarAlturas é usada na função reduce para somar as alturas das plantas e calcular a soma total. A função calcularMedia recebe uma lista de valores e retorna a média desses valores. A constante mediaDeAlturaDasPlantas é criada a partir da composição das funções obterAlturas, somarAlturas e calcularMedia. Ela obtém as alturas das plantas usando a função obterAlturas, soma essas alturas usando a função somarAlturas e calcula a média usando a função calcularMedia.

## Recursão

Aqui estão alguns exemplos de como a recursão pode ser usada com o tema Grécia moderna:

**Exemplo 1: Recursão para obter a soma dos números de 1 a n**

*const* soma = *n* *=>* {

    if (n === 1) {

        return 1;

    } else {

        return n + soma(n - 1);

    }

};

console.log(soma(5)); // 15

Neste exemplo, a função soma recebe um número n e retorna a soma dos números de 1 até n. A função usa a recursão para somar os números de forma iterativa. Se n for igual a 1, a função retorna 1. Caso contrário, ela soma n ao resultado da chamada recursiva da função soma com o argumento n - 1.

**Exemplo 2: Recursão para obter o fatorial de um número**

*const* fatorial = *n* *=>* {

    if (n === 0 || n === 1) {

        return 1;

    } else {

        return n \* fatorial(n - 1);

    }

};

console.log(fatorial(5)); // 120

Neste exemplo, a função fatorial recebe um número n e retorna o fatorial desse número. A função usa a recursão para calcular o fatorial de forma iterativa. Se n for igual a 0 ou 1, a função retorna 1. Caso contrário, ela retorna n multiplicado pelo resultado da chamada recursiva da função fatorial com o argumento n - 1.

**Exemplo 3: Recursão para obter o enésimo termo da sequência de Fibonacci**

*const* fibonacci = *n* *=>* {

    if (n === 0 || n === 1) {

        return n;

    } else {

        return fibonacci(n - 1) + fibonacci(n - 2);

    }

};

console.log(fibonacci(6)); // 8

Neste exemplo, a função fibonacci recebe um número n e retorna o enésimo termo da sequência de Fibonacci. A função usa a recursão para calcular o termo de forma iterativa. Se n for igual a 0 ou 1, a função retorna n. Caso contrário, ela retorna a soma dos resultados da chamada recursiva da função fibonacci com os argumentos n - 1 e n - 2.

## Mais sobre recursão na programação funcional em javascript

A recursão é um conceito fundamental na programação funcional e é muito comum em JavaScript. A recursão é o processo de resolver um problema dividindo-o em subproblemas menores, de forma que a solução do problema original possa ser encontrada combinando as soluções dos subproblemas. Em JavaScript, a recursão é usada para implementar muitos algoritmos comuns, como a busca em profundidade em uma árvore, o cálculo do fatorial e o algoritmo de ordenação quicksort.

Aqui estão algns exemplos de como a recursão pode ser usada em JavaScript:

**Exemplo 1: Recursão para obter a soma dos números de 1 a n**

*const* soma = *n* *=>* {

    if (n === 1) {

        return 1;

    } else {

        return n + soma(n - 1);

    }

};

console.log(soma(5)); // 15

Neste exemplo, a função soma recebe um número n e retorna a soma dos números de 1 até n. A função usa a recursão para somar os números de forma iterativa. Se n for igual a 1, a função retorna 1. Caso contrário, ela soma n ao resultado da chamada recursiva da função soma com o argumento n - 1.

**Exemplo 2: Recursão para obter o fatorial de um número**

*const* fatorial = *n* *=>* {

    if (n === 0 || n === 1) {

        return 1;

    } else {

        return n \* fatorial(n - 1);

    }

};

console.log(fatorial(5)); // 120

Neste exemplo, a função fatorial recebe um número n e retorna o fatorial desse número. A função usa a recursão para calcular o fatorial de forma iterativa. Se n for igual a 0 ou 1, a função retorna 1. Caso contrário, ela retorna n multiplicado pelo resultado da chamada recursiva da função fatorial com o argumento n - 1.

**Exemplo 3: Recursão para obter o enésimo termo da sequência de Fibonacci**

*const* fibonacci = *n* *=>* {

    if (n === 0 || n === 1) {

        return n;

    } else {

        return fibonacci(n - 1) + fibonacci(n - 2);

    }

};

console.log(fibonacci(6)); // 8

Neste exemplo, a função fibonacci recebe um número n e retorna o enésimo termo da sequência de Fibonacci. A função usa a recursão para calcular o termo de forma iterativa. Se n for igual a 0 ou 1, a função retorna n. Caso contrário, ela retorna a soma dos resultados da chamada recursiva da função fibonacci com os argumentos n - 1 e n - 2.

A recursão pode ser usada com strings e objetos em JavaScript. A recursão é uma técnica poderosa que pode ser usada para processar dados em muitos formatos diferentes, incluindo strings e objetos. Aqui estão alguns exemplos de como a recursão pode ser usada com esses tipos de dados em JavaScript:

**Exemplo 1: Recursão para percorrer uma string**

*const* percorrerString = (*str*, *i* = 0) *=>* {

    if (i >= str.length) {

        return;

    }

    console.log(str[i]);

    percorrerString(str, i + 1);

};

percorrerString("Hello, world!"); // H e l l o ,   w o r l d !

Neste exemplo, a função percorrerString recebe uma string str e um índice i opcional que indica a posição atual na string. A função usa a recursão para percorrer a string de forma iterativa. Se i for maior ou igual ao comprimento da string, a função retorna. Caso contrário, ela exibe o caractere na posição i e chama a si mesma com o argumento i + 1.

**Exemplo 2: Recursão para percorrer um objeto**

*const* percorrerObjeto = (*obj*, *prefixo* = "") *=>* {

    for (*const* chave in obj) {

*const* valor = obj[chave];

        if (typeof valor === "object") {

            percorrerObjeto(valor, prefixo + chave + ".");

        } else {

            console.log(prefixo + chave + " = " + valor);

        }

    }

};

*const* pessoa = {

    nome: "João",

    idade: 30,

    endereço: {

        rua: "Rua A",

        número: 123,

        cidade: "São Paulo",

        estado: "SP"

    }

};

percorrerObjeto(pessoa);

  // nome = João

  // idade = 30

  // endereço.rua = Rua A

  // endereço.número = 123

  // endereço.cidade = São Paulo

  // endereço.estado = SP

Neste exemplo, a função percorrerObjeto recebe um objeto obj e um prefixo prefixo opcional que é usado para construir o nome das chaves. A função usa a recursão para percorrer o objeto de forma iterativa. Se o valor da chave for outro objeto, a função chama a si mesma com o objeto e o prefixo atualizado. Caso contrário, ela exibe o nome da chave e o valor correspondente.

Aqui está um exemplo de recursão que utiliza o tema Grécia:

*const* paises = {

    grecia: {

        capital: "Atenas",

        cidades: {

            atenas: {

                populacao: 664046,

                monumentos: ["Acrópole", "Partenon"]

            },

            thessaloniki: {

                populacao: 385406,

                monumentos: ["Arco de Galério", "Igreja de São Demétrio"]

            },

            patras: {

                populacao: 168034,

                monumentos: ["Castelo de Patras", "Ponte Rio-Antirrio"]

            }

        }

    }

};

*const* listarMonumentos = (*paises*) *=>* {

    for (*const* pais in paises) {

        console.log(`Monumentos na ${pais}:`);

*const* cidades = paises[pais].cidades;

        listarMonumentosRecursivo(cidades);

    }

};

*const* listarMonumentosRecursivo = (*cidades*) *=>* {

    for (*const* cidade in cidades) {

*const* monumentos = cidades[cidade].monumentos;

        console.log(`- ${cidade}:`);

        listarMonumentosRecursivoAux(monumentos);

    }

};

*const* listarMonumentosRecursivoAux = (*monumentos*, *i* = 0) *=>* {

    if (i >= monumentos.length) {

        return;

    }

    console.log(`  - ${monumentos[i]}`);

    listarMonumentosRecursivoAux(monumentos, i + 1);

};

listarMonumentos(paises);

Neste exemplo, temos um objeto paises que representa a Grécia e suas cidades. A função listarMonumentos usa um loop for...in para percorrer cada país e chama a função listarMonumentosRecursivo para listar os monumentos de cada cidade. A função listarMonumentosRecursivo também usa um loop for...in para percorrer cada cidade e chama a função listarMonumentosRecursivoAux para listar os monumentos de cada cidade. A função listarMonumentosRecursivoAux usa a recursão para listar os monumentos de forma iterativa. Se i for maior ou igual ao comprimento da matriz de monumentos, a função retorna. Caso contrário, ela exibe o monumento na posição i e chama a si mesma com o argumento i + 1.

Há muitas outras coisas interessantes a se aprender sobre a programação funcional em JavaScript! Aqui estão algumas outras características e conceitos que você pode querer explorar:

**High-order functions:** funções que podem receber outras funções como argumentos ou retornar funções como resultado. Isso é muito útil para criar funções mais genéricas e reutilizáveis.

**Closures:** funções que capturam e mantêm o escopo em que foram criadas, mesmo após a função externa ter retornado. Isso permite que você crie funções com estado e comportamento persistentes.

**Currying:** uma técnica que transforma uma função de múltiplos argumentos em uma sequência de funções que levam apenas um argumento. Isso é útil para criar funções mais especializadas e flexíveis.

**Memoization:** uma técnica que armazena em cache os resultados de uma função para evitar a repetição desnecessária de cálculos. Isso é útil para melhorar o desempenho de funções que levam muito tempo para executar.

**Funções assíncronas:** funções que retornam uma promessa (Promise) ou usam a palavra-chave async para permitir que o código continue a ser executado enquanto a função está sendo executada em segundo plano. Isso é útil para evitar bloqueios de thread e criar código assíncrono mais fácil de ler e manter.

Esses são apenas alguns exemplos, e há muito mais a aprender sobre programação funcional em JavaScript. A chave é experimentar e praticar com as várias técnicas e padrões disponíveis para encontrar o estilo de programação que melhor se adapta ao seu projeto.

Veja alguns exemplos de códigos para cada um dos conceitos mencionados acima usando o tema Antártida:

### High-order functions:

**// Exemplo 1: uma função que recebe uma função de callback e a aplica a cada elemento de um array**

*function* aplicarFuncao(*callback*, *array*) {

    return array.map(callback);

}

*const* animais = ['pinguim', 'baleia', 'foca'];

**// Aplicar a função maiúscula a cada elemento do array de animais**

*const* animaisMaiuscula = aplicarFuncao((*animal*) *=>* animal.toUpperCase(), animais);

console.log(animaisMaiuscula); // ['PINGUIM', 'BALEIA', 'FOCA']

**// Exemplo 2: uma função que retorna uma função de soma que recebe um número constante**

*function* soma(*numero*) {

    return (*outroNumero*) *=>* numero + outroNumero;

}

**// Criar funções de soma com números constantes**

*const* somaCom5 = soma(5);

*const* somaCom10 = soma(10);

// Usar as funções de soma com diferentes números

console.log(somaCom5(3)); // 8

console.log(somaCom10(7)); // 17

### Closures:

**// Exemplo 1: uma função que cria um contador persistente usando um closure**

*function* criarContador() {

*let* contador = 0;

    return () *=>* {

        contador++;

        console.log(contador);

    };

}

*const* contador1 = criarContador();

contador1(); // 1

contador1(); // 2

contador1(); // 3

*const* contador2 = criarContador();

contador2(); // 1

contador2(); // 2

**// Exemplo 2: uma função que cria um objeto com um método persistente usando um closure**

*function* criarObjeto() {

*let* valor = 0;

    return {

        getValor: () *=>* valor,

        setValor: (*novoValor*) *=>* {

            valor = novoValor;

            console.log(`Valor atualizado para ${valor}`);

        }

    };

}

*const* objeto1 = criarObjeto();

console.log(objeto1.getValor()); // 0

objeto1.setValor(10); // Valor atualizado para 10

console.log(objeto1.getValor()); // 10

### Currying:

**// Exemplo 1: uma função que faz curry em uma função de soma**

function soma(a, b) {

return a + b;

}

function currySoma(a) {

return (b) => soma(a, b);

}

// Usar a função currySoma para criar funções de soma com um número constante

const somaCom5 = currySoma(5);

const somaCom10 = currySoma(10);

console.log(somaCom5(3)); // 8

console.log(somaCom10(7)); // 17

// Exemplo 2: uma função que faz curry em uma função de formatação de string

function formatarString(texto, prefixo, sufixo) {

return `${prefixo}${texto}${sufixo}`;

}

function curryFormatarString(prefixo) {

return (sufixo) => (texto) => formatarString(texto, prefixo, sufixo);

}

// Usar a função curryFormatarString para criar funções de formatação de string com prefixo e sufixo constantes

const formatarComParenteses = curryFormatarString('(')(')');

const formatarComColchet

Rogério Soares (Merovingio)

continue de onde parou const formatarComColchet

Claro, podemos continuar de onde paramos. Abaixo está o código da função formatarComColchetes que formata um array de strings com colchetes:

javascript

Copy code

function formatarComColchetes(arr) {

if (arr.length === 0) {

return "[]";

} else {

const [primeiro, ...resto] = arr;

const corpo = resto.length > 0 ? `, ${formatarComColchetes(resto)}` : "";

return `[${primeiro}${corpo}]`;

}

}

E agora vamos continuar com a próxima função.

contarVogais

Essa função recebe uma string como entrada e retorna o número de vogais que ela contém. Para implementá-la, podemos utilizar a função reduce para iterar sobre os caracteres da string e contar as vogais. O código ficaria assim:

javascript

Copy code

function contarVogais(str) {

const vogais = ["a", "e", "i", "o", "u"];

return str

.toLowerCase()

.split("")

.reduce((contador, caractere) => {

if (vogais.includes(caractere)) {

return contador + 1;

} else {

return contador;

}

}, 0);

}

Nesse exemplo, a função toLowerCase é usada para transformar todas as letras da string em minúsculas, e a função split é usada para transformar a string em um array de caracteres. Depois disso, a função reduce é utilizada para iterar sobre o array de caracteres e contar as vogais. O segundo parâmetro da função reduce é o valor inicial do contador, que começa em 0.

Rogério Soares (Merovingio)

continue os exemplos com Memoization e Funções assíncronas

Claro, vamos continuar com mais dois exemplos:

Memoization

Memoization é uma técnica utilizada para otimizar funções que realizam cálculos custosos. A ideia é armazenar o resultado da função para um dado conjunto de argumentos, de forma que, caso a função seja chamada novamente com os mesmos argumentos, o resultado já esteja armazenado e possa ser retornado imediatamente sem precisar calcular novamente. Em JavaScript, podemos implementar memoization utilizando um objeto para armazenar os resultados das chamadas anteriores da função.

Abaixo está um exemplo de função que implementa memoization para calcular o fatorial de um número:

javascript

Copy code

function memoize(fn) {

const cache = {};

return function(...args) {

const strArgs = JSON.stringify(args);

if (cache[strArgs] === undefined) {

cache[strArgs] = fn.apply(null, args);

}

return cache[strArgs];

};

}

function fatorial(n) {

if (n === 0) {

return 1;

} else {

return n \* fatorial(n - 1);

}

}

const fatorialMemoizado = memoize(fatorial);

console.log(fatorialMemoizado(5)); // 120

console.log(fatorialMemoizado(5)); // 120 (o resultado é retornado do cache)

Nesse exemplo, a função memoize recebe uma função fn como parâmetro e retorna uma nova função que implementa memoization. Essa nova função utiliza um objeto cache para armazenar os resultados das chamadas anteriores da função fn. O objeto cache é indexado pelos argumentos da função fn, que são transformados em uma string com a função JSON.stringify. Se o resultado para aqueles argumentos ainda não está no cache, a função fn é chamada para calcular o resultado, que é armazenado no cache. Se o resultado já está no cache, ele é retornado imediatamente.

Funções assíncronas

Funções assíncronas são funções que realizam operações assíncronas, como chamadas a APIs ou operações de I/O, e retornam uma promessa. Em JavaScript, podemos utilizar a sintaxe async/await para trabalhar com funções assíncronas de forma síncrona e mais legível.

Abaixo está um exemplo de função que utiliza async/await para fazer uma chamada a uma API que retorna a temperatura atual em uma cidade:

javascript

Copy code

async function obterTemperaturaAtual(cidade) {

const url = `https://api.openweathermap.org/data/2.5/weather?q=${cidade}&appid=${API\_KEY}&units=metric`;

const response = await fetch(url);

const data = await response.json();

return data.main.temp;

}

obterTemperaturaAtual("São Paulo")

.then(temp => console.log(`A temperatura atual em São Paulo é ${temp}°C.`))

.catch(err => console.error(err));

Nesse exemplo, a função obterTemperaturaAtual utiliza a palavra-chave async para indicar que ela é assíncrona e pode utilizar a sintaxe await. Dentro da função, é feita uma chamada à API do OpenWeatherMap para obter a temperatura atual na cidade passada como parâmetro. Essa chamada é feita utilizando a função fetch, que retorna uma promessa que é resolvida com a resposta da API

Rogério Soares (Merovingio)

continue de onde parou resolvida com a resposta da API

Ok, continuando a partir do ponto em que paramos:

Assumindo que já temos a função buscarDadosApi que faz uma requisição para a API, podemos agora utilizar a memoização para otimizar o desempenho da nossa aplicação.

A memoização é uma técnica de otimização que consiste em armazenar o resultado de uma função para evitar que ela seja executada novamente com os mesmos parâmetros. Para isso, podemos criar uma cache que armazena os resultados das chamadas da função. Se a mesma chamada for feita novamente com os mesmos parâmetros, a função retorna o resultado armazenado na cache em vez de executar a chamada novamente.

Veja um exemplo de como utilizar a memoização em uma função assíncrona que busca informações de um usuário a partir de um ID:

javascript

Copy code

const buscarUsuario = async (id) => {

if (!cache[id]) {

const dados = await buscarDadosApi(`/usuarios/${id}`);

cache[id] = dados;

}

return cache[id];

};

const cache = {};

buscarUsuario(1).then((dados) => console.log(dados));

// faz a chamada para a API e armazena o resultado na cache

buscarUsuario(1).then((dados) => console.log(dados));

// retorna o resultado armazenado na cache em vez de fazer a chamada novamente

Neste exemplo, criamos uma função buscarUsuario que recebe um ID de usuário como parâmetro. A função verifica se os dados do usuário já estão armazenados na cache. Se sim, retorna o valor armazenado na cache. Se não, faz a chamada para a API para buscar os dados do usuário e armazena o resultado na cache.

Ao chamar a função duas vezes com o mesmo ID, a segunda chamada não faz a chamada para a API novamente, mas retorna o valor armazenado na cache.

Além da memoização, outra técnica que pode ser útil em funções assíncronas é a utilização de funções de callback ou Promises para lidar com operações assíncronas.

Veja um exemplo de como utilizar uma função de callback para lidar com a resposta de uma chamada assíncrona:

javascript

Copy code

const buscarDadosApi = (url, callback) => {

// faz a chamada para a API

fetch(url)

.then((response) => response.json())

.then((data) => callback(data))

.catch((error) => console.error(error));

};

const url = "https://jsonplaceholder.typicode.com/posts/1";

buscarDadosApi(url, (data) => console.log(data));

Neste exemplo, criamos uma função buscarDadosApi que recebe uma URL e uma função de callback como parâmetros. A função faz a chamada para a API utilizando o método fetch e, quando recebe a resposta, converte o conteúdo para JSON e passa para a função de callback.

Ao chamar a função buscarDadosApi, passamos a URL e uma função de callback que imprime os dados no console.

A utilização de funções de callback ou Promises pode ajudar a tornar o código mais legível e modular, separando as operações assíncronas do resto do código.

Regenerate response

Regenera

# Closures e escopo em JavaScript

Closures e escopo em JavaScript

Em JavaScript, um closure (fechamento) é uma função que lembra do ambiente no qual foi criada, mesmo após essa função ter saído do escopo onde foi declarada. Isso significa que a função pode acessar e manipular variáveis que estão fora do seu próprio escopo.

O escopo em JavaScript é definido pela forma como as variáveis são declaradas. Se uma variável é declarada dentro de uma função, ela é uma variável local e só pode ser acessada dentro dessa função. Se uma variável é declarada fora de uma função, ela é uma variável global e pode ser acessada por qualquer função em todo o código.

Quando uma função é criada dentro de outra função em JavaScript, a função interna tem acesso às variáveis da função externa, mesmo após a função externa ter sido concluída. Isso é possível porque a função interna "fecha" em torno das variáveis da função externa, criando um closure.

Por exemplo:

scss

Copy code

function outer() {

var x = 10;

function inner() {

console.log(x);

}

return inner;

}

var closure = outer();

closure(); // retorna 10

Neste exemplo, a função outer cria uma variável x e uma função inner. A função inner é retornada pela função outer e atribuída à variável closure. Quando a função closure é executada, ela imprime o valor de x, que é 10, mesmo que a função outer já tenha retornado e a variável x não esteja mais no escopo.

Os closures são frequentemente usados em JavaScript para criar funções privadas, que são funções que não podem ser acessadas fora do seu próprio escopo. Eles também podem ser usados para armazenar informações entre as chamadas de função ou para criar funções de retorno de chamada personalizadas.

Rogério Soares (Merovingio)

cite mais exemplos usando o tema oceano

Claro, aqui estão alguns exemplos usando o tema oceano:

Ecossistemas marinhos: O oceano é o lar de muitos ecossistemas, desde recifes de coral e pradarias marinhas até estuários e zonas abissais. Esses ecossistemas abrigam uma incrível diversidade de vida marinha, incluindo peixes, mamíferos marinhos, crustáceos, moluscos e muito mais.

Aquecimento global e acidificação dos oceanos: O aquec

Rogério Soares (Merovingio)

cite mais exemplos usando o tema oceano de closure (fechamento) javascript e escopo

Com certeza, aqui estão alguns exemplos usando o tema oceano de closure e escopo em JavaScript:

Closures e animais marinhos: Imagine que você está construindo um aplicativo para uma organização de conservação marinha que rastreia populações de animais marinhos. Você pode usar closures para criar funções que armazenam informações sobre cada espécie de animal, como a quantidade de indivíduos em uma população, as áreas geográficas em que eles são encontrados e os desafios que enfrentam em termos de conservação.

Escopo e medição de dados oceânicos: Imagine que você está criando um aplicativo para coletar e analisar dados oceânicos, como a temperatura da água, a salinidade e a acidez. Você pode usar escopos locais e globais para garantir que as informações coletadas em diferentes locais sejam armazenadas de forma segura e acessível. Além disso, você pode usar closures para criar funções de retorno de chamada que sejam acionadas quando novos dados são coletados.

Closures e roteamento de navios: Imagine que você está criando um aplicativo para uma empresa de transporte marítimo que precisa rastrear a localização de seus navios em tempo real. Você pode usar closures para criar funções que armazenam informações sobre cada navio, como sua posição atual e a velocidade em que está navegando. Além disso, você pode usar escopos locais e globais para garantir que os dados do navio sejam armazenados de forma segura e acessível a outras partes do aplicativo que precisam dessas informações.

Rogério Soares (Merovingio)

faça os códigos em javascript do que foi citado

Claro, aqui estão alguns exemplos de código em JavaScript baseados nos cenários que mencionei anteriormente:

Closures e animais marinhos:

javascript

Copy code

function criarAnimal(nome, populacao, habitat, desafios) {

var animal = {

nome: nome,

populacao: populacao,

habitat: habitat,

desafios: desafios

};

function atualizarPopulacao(novaPopulacao) {

animal.populacao = novaPopulacao;

console.log(animal.nome + ' agora tem uma população de ' + novaPopulacao);

}

return {

animal: animal,

atualizarPopulacao: atualizarPopulacao

};

}

var baleiaAzul = criarAnimal('Baleia-azul', 25000, 'Oceano', ['Caça e pesca excessivas']);

baleiaAzul.atualizarPopulacao(23000);

Neste exemplo, a função criarAnimal cria um objeto animal com várias propriedades e uma função atualizarPopulacao que atualiza a população do animal e exibe uma mensagem de confirmação no console. A função retorna um objeto que inclui o objeto animal e a função atualizarPopulacao. Quando a função atualizarPopulacao é chamada no final, ela atualiza a população da baleia-azul para 23.000 e exibe a mensagem 'Baleia-azul agora tem uma população de 23000' no console.

Escopo e medição de dados oceânicos:

javascript

Copy code

var dadosOceanicos = {

temperatura: 18,

salinidade: 35,

acidez: 7.8

};

function coletarDados() {

var temperatura = prompt('Insira a temperatura da água:');

var salinidade = prompt('Insira a salinidade da água:');

var acidez = prompt('Insira o nível de acidez da água:');

function exibirDados() {

console.log('Dados coletados: ');

console.log('Temperatura: ' + temperatura);

console.log('Salinidade: ' + salinidade);

console.log('Acidez: ' + acidez);

}

dadosOceanicos.temperatura = temperatura;

dadosOceanicos.salinidade = salinidade;

dadosOceanicos.acidez = acidez;

exibirDados();

}

coletarDados();

Neste exemplo, a função coletarDados solicita ao usuário que insira a temperatura, salinidade e acidez da água. Em seguida, ele atualiza os valores de dadosOceanicos com essas informações e chama a função exibirDados, que exibe as informações no console. Como as variáveis temperatura, salinidade e acidez são declaradas dentro da função coletarDados, elas têm escopo local e não podem ser acessadas fora dessa função.

Closures e roteamento de navios:

javascript

Copy code

function criarNavio(nome, latitudeInicial, longitudeInicial, velocidadeInicial) {

var navio = {

nome: nome,

latitude: latitudeInicial,

longitude: longitudeInicial,

velocidade: velocidadeInicial

};

function atualizarLocalizacao(latitude, longitude) {

navio

Rogério Soares (Merovingio)

continue de onde parou

arduino

Copy code

navio.latitude = latitude;

navio.longitude = longitude;

console.log(navio.nome + ' atualizou sua localização para ' + latitude + ', ' + longitude);

}

function atualizarVelocidade(novaVelocidade) {

navio.velocidade = novaVelocidade;

console.log(navio.nome + ' agora está viajando a ' + novaVelocidade + ' nós');

}

function exibirLocalizacao() {

console.log(navio.nome + ' está atualmente em ' + navio.latitude + ', ' + navio.longitude);

}

return {

atualizarLocalizacao: atualizarLocalizacao,

atualizarVelocidade: atualizarVelocidade,

exibirLocalizacao: exibirLocalizacao

};

}

var navio1 = criarNavio('Rainha dos Mares', -22.967, -43.18, 12);

var navio2 = criarNavio('Mar de Esperança', 33.75, -118.17, 8);

navio1.atualizarLocalizacao(-23.8, -45.3);

navio2.atualizarVelocidade(10);

navio1.exibirLocalizacao();

kotlin

Copy code

Neste exemplo, a função `criarNavio` cria um objeto `navio` com várias propriedades e três funções que atualizam a localização, a velocidade e exibem a localização do navio. A função retorna um objeto que inclui essas três funções. Quando os métodos são chamados para atualizar a localização e a velocidade dos navios, eles atualizam as propriedades correspondentes e exibem uma mensagem no console confirmando a atualização. No final, a função `exibirLocalizacao` é chamada para exibir a localização atual do navio 1.

Rogério Soares (Merovingio)

Mais exemplos usando o tema aquecimento global

Claro! Aqui estão alguns exemplos adicionais que usam o tema aquecimento global em JavaScript:

Função que calcula a temperatura média anual:

css

Copy code

function calcularTemperaturaMedia(temperaturasAnuais) {

var totalTemperaturas = 0;

for (var i = 0; i < temperaturasAnuais.length; i++) {

totalTemperaturas += temperaturasAnuais[i];

}

var temperaturaMedia = totalTemperaturas / temperaturasAnuais.length;

return temperaturaMedia;

}

var temperaturasAnuais = [12, 15, 18, 20, 22, 24, 26, 28, 30, 32];

var temperaturaMediaAnual = calcularTemperaturaMedia(temperaturasAnuais);

console.log('A temperatura média anual é de ' + temperaturaMediaAnual + ' graus Celsius.');

Neste exemplo, a função calcularTemperaturaMedia recebe um array de temperaturas anuais e calcula a temperatura média anual, retornando esse valor. A função é chamada passando um array de temperaturas anuais, e o valor da temperatura média anual é exibido no console.

Objeto que representa uma cidade com dados sobre o clima:

javascript

Copy code

var cidade = {

nome: 'São Paulo',

populacao: 12325232,

temperaturaMediaAnual: 20.5,

niveisPoluicao: {

co2: 245,

metano: 15,

oxidoNitroso: 8

},

exibirDadosClimaticos: function() {

console.log('Dados climáticos de ' + this.nome + ':');

console.log('- Temperatura média anual: ' + this.temperaturaMediaAnual + ' graus Celsius');

console.log('- Níveis de poluição:');

console.log(' - CO2: ' + this.niveisPoluicao.co2 + ' partes por milhão');

console.log(' - Metano: ' + this.niveisPoluicao.metano + ' partes por bilhão');

console.log(' - Óxido nitroso: ' + this.niveisPoluicao.oxidoNitroso + ' partes por milhão');

}

};

cidade.exibirDadosClimaticos();

Neste exemplo, um objeto cidade é criado com várias propriedades, incluindo temperaturaMediaAnual e niveisPoluicao, que contêm dados sobre o clima. O objeto também inclui um método exibirDadosClimaticos, que exibe as informações climáticas da cidade no console. O método é chamado no final do código para exibir os dados climáticos da cidade de São Paulo.

Claro, vou fornecer alguns exemplos de closures em JavaScript/ECMAScript 2021:

Exemplo básico de closure:

javascript

Copy code

function outerFunction() {

let outerVariable = 'Hello';

function innerFunction() {

console.log(outerVariable);

}

return innerFunction;

}

const innerFunc = outerFunction();

innerFunc(); // Output: 'Hello'

Neste exemplo, a função outerFunction cria uma variável outerVariable e uma função innerFunction que acessa a variável outerVariable. A função outerFunction retorna a innerFunction e, em seguida, é atribuída a uma variável chamada innerFunc. Quando innerFunc é invocada, ela acessa a variável outerVariable declarada na função outerFunction, graças à closure.

Exemplo de closure com parâmetros:

javascript

Copy code

function greeting(name) {

return function() {

console.log(`Hello, ${name}!`);

}

}

const greetBob = greeting('Bob');

greetBob(); // Output: 'Hello, Bob!'

Neste exemplo, a função greeting retorna uma nova função que recebe um parâmetro name e retorna uma string de saudação com o nome fornecido. A nova função criada pela greeting é atribuída a uma variável greetBob, que pode ser invocada mais tarde, produzindo a saudação correta graças à closure.

Exemplo de closure com modificação de variável:

scss

Copy code

function counter() {

let count = 0;

return function() {

count++;

console.log(`Count: ${count}`);

}

}

const incrementCounter = counter();

incrementCounter(); // Output: 'Count: 1'

incrementCounter(); // Output: 'Count: 2'

Neste exemplo, a função counter retorna uma nova função que incrementa uma variável count e exibe seu valor atual. A nova função é atribuída a uma variável incrementCounter, que é invocada duas vezes para demonstrar que a variável count está sendo modificada corretamente graças à closure.

# Tipos de dados avançados em JavaScript (Map, Set, WeakMap, WeakSet)

Tipos de dados avançados em JavaScript (Map, Set, WeakMap, WeakSet)

JavaScript oferece algumas estruturas de dados avançadas que podem ser úteis em certas situações. Aqui estão quatro delas:

Map:

Map é uma estrutura de dados que mapeia chaves a valores. Em outras palavras, é uma coleção de pares de chave-valor. As chaves podem ser de qualquer tipo de dado, incluindo objetos e funções. É semelhante a um objeto, mas com algumas diferenças importantes, como poder usar qualquer tipo de dados para chaves, ter um tamanho acessível através da propriedade size, e possuir uma ordem de inserção previsível.

Exemplo de uso:

c

Copy code

const map = new Map();

map.set('a', 1);

map.set('b', 2);

map.set('c', 3);

console.log(map.get('a')); // 1

console.log(map.size); // 3

console.log(map.has('d')); // false

Set:

Set é uma estrutura de dados que armazena valores únicos de qualquer tipo de dado. É semelhante a um array, mas com a diferença de que não permite valores duplicados. Também possui um tamanho acessível através da propriedade size.

Exemplo de uso:

csharp

Copy code

const set = new Set();

set.add(1);

set.add(2);

set.add(3);

set.add(1);

console.log(set.size); // 3

console.log(set.has(2)); // true

WeakMap:

WeakMap é uma versão fraca (ou frágil) do Map. A principal diferença é que as chaves devem ser objetos, e esses objetos só são mantidos enquanto ainda existirem referências a eles em outras partes do código. Quando um objeto que é uma chave de um WeakMap não é mais referenciado, ele é removido automaticamente do mapa.

Exemplo de uso:

csharp

Copy code

const weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, 'some value');

console.log(weakMap.get(obj)); // 'some value'

obj = null;

// O objeto é automaticamente removido do WeakMap

WeakSet:

WeakSet é uma versão fraca (ou frágil) do Set. Assim como o WeakMap, ele só pode armazenar objetos e só mantém esses objetos enquanto ainda existirem referências a eles em outras partes do código. Quando um objeto é removido da memória, ele é automaticamente removido do WeakSet.

Exemplo de uso:

javascript

Copy code

const weakSet = new WeakSet();

let obj = {};

weakSet.add(obj);

console.log(weakSet.has(obj)); // true

obj = null;

// O objeto é automaticamente removido do WeakSet

Em resumo, Map e Set são estruturas de dados para armazenamento de informações, enquanto WeakMap e WeakSet são versões frágeis dessas estruturas que podem ser úteis em certas situações em que você precisa garantir que os objetos armazenados não sejam mantidos na memória quando não são mais necessários.

Rogério Soares (Merovingio)

cite mais exemplos de cada um usando o tema matemática avançada

Claro, aqui estão alguns exemplos de como as estruturas de dados avançadas em JavaScript podem ser úteis em tarefas de matemática avançada:

Map:

Armazenamento de coordenadas em um mapa: você pode usar um Map para armazenar pares de coordenadas (x, y) e associar cada um a um valor. Por exemplo, você pode armazenar os pontos de um gráfico de função e associá-los a valores calculados para a função nesses pontos.

php

Copy code

const graphPoints = new Map();

// Adicionando pontos (x, y) e seus valores

graphPoints.set({x: 1, y: 2}, 3.14);

graphPoints.set({x: 2, y: 3}, 2.72);

graphPoints.set({x: 3, y: 4}, 1.62);

console.log(graphPoints.get({x: 2, y: 3})); // 2.72

Set:

Armazenamento de números primos: você pode usar um Set para armazenar números primos. Como um Set só armazena valores únicos, você pode garantir que cada número primo só é armazenado uma vez.

javascript

Copy code

const primeNumbers = new Set();

function isPrime(n) {

if (n <= 1) return false;

for (let i = 2; i <= Math.sqrt(n); i++) {

if (n % i === 0) return false;

}

return true;

}

for (let i = 1; i <= 100; i++) {

if (isPrime(i)) {

primeNumbers.add(i);

}

}

console.log(primeNumbers.has(7)); // true

console.log(primeNumbers.has(10)); // false

WeakMap:

Armazenamento de funções em um mapa: você pode usar um WeakMap para armazenar funções e associá-las a objetos. Isso pode ser útil em tarefas de otimização de código, onde você pode querer armazenar funções que dependem de objetos grandes que não precisam ser mantidos na memória por muito tempo.

javascript

Copy code

const functionMap = new WeakMap();

const obj1 = {x: 1, y: 2};

const obj2 = {x: 2, y: 3};

function calcDistance(obj1, obj2) {

const dx = obj2.x - obj1.x;

const dy = obj2.y - obj1.y;

return Math.sqrt(dx \* dx + dy \* dy);

}

functionMap.set(obj1, calcDistance);

console.log(functionMap.get(obj1)(obj1, obj2)); // 1.4142135623730951

obj1 = null;

// obj1 e a função associada são automaticamente removidos do WeakMap

WeakSet:

Armazenamento de referências de matriz em um conjunto: você pode usar um WeakSet para armazenar referências de matrizes que são usadas temporariamente em tarefas de cálculo de matrizes. Como as referências de matriz não são mais necessárias após o cálculo, elas podem ser removidas da memória automaticamente pelo WeakSet.

csharp

Copy code

const matrixRefs = new WeakSet();

function multiplyMatrices(matrixA, matrixB) {

const resultMatrix = /\* cálculo de matriz \*/;

matrixRefs.add(matrixA);

matrixRefs.add(matrixB);

return resultMatrix;

}

const

Rogério Soares (Merovingio)

continue

// criando matrizes para teste

const matrix1 = [[1, 2], [3, 4]];

const matrix2 = [[5, 6], [7, 8]];

const resultMatrix = multiplyMatrices(matrix1, matrix2);

console.log(resultMatrix); // exibe a matriz resultante

// matrix1 e matrix2 são removidos automaticamente do WeakSet

css

Copy code

Esses são apenas alguns exemplos de como as estruturas de dados avançadas em JavaScript podem ser usadas em tarefas de matemática avançada. Dependendo do problema que você está resolvendo, essas estruturas de dados podem ser muito úteis em outras áreas, como análise de dados, aprendizado de máquina, etc.

# Desenvolvimento de aplicações SPA (Single Page Application)

As aplicações SPA (Single Page Application) são aplicações web que carregam uma única página HTML e atualizam dinamicamente o conteúdo dessa página, em vez de carregar novas páginas em cada interação do usuário. Essa abordagem permite uma experiência de usuário mais fluida e responsiva, com menos interrupções e tempo de carregamento reduzido.

Para desenvolver uma aplicação SPA, geralmente é necessário seguir algumas práticas recomendadas:

Escolher um framework de JavaScript: Um framework de JavaScript, como o React, Angular ou Vue.js, fornece uma estrutura e bibliotecas de suporte para desenvolver uma aplicação SPA.

Implementar um roteador: Um roteador é responsável por gerenciar as rotas da aplicação e renderizar o componente apropriado para cada rota. O roteador pode ser construído usando uma biblioteca de terceiros, como o react-router ou vue-router.

Gerenciar o estado da aplicação: Para criar uma aplicação SPA escalável, é essencial gerenciar o estado da aplicação de forma eficiente. Isso pode ser feito usando uma biblioteca de gerenciamento de estado, como o Redux ou Vuex.

Otimizar o desempenho: Uma aplicação SPA pode ter problemas de desempenho, como tempos de carregamento longos ou consumo excessivo de memória. É importante otimizar a aplicação para garantir um bom desempenho. Isso pode incluir técnicas como lazy loading, cache de recursos, redução do tamanho dos arquivos e minimização de solicitações HTTP.

Testar e depurar: Como em qualquer desenvolvimento de software, é importante testar e depurar a aplicação SPA para garantir que ela seja confiável e livre de bugs. Isso pode ser feito usando ferramentas de teste de unidade, testes de integração e depuração em tempo real.

Em resumo, o desenvolvimento de uma aplicação SPA requer a escolha adequada de um framework de JavaScript, implementação de um roteador, gerenciamento de estado, otimização de desempenho, teste e depuração. Com essas práticas recomendadas em mente, é possível criar uma aplicação SPA eficiente e escalável.

Rogério Soares (Merovingio)

Cite um exemplo bem complexo de código de aplicações SPA (Single Page Application)

Uma aplicação SPA complexa pode ser difícil de apresentar de forma completa em um exemplo, mas posso citar um exemplo comum de código para uma aplicação SPA usando o framework React e outras bibliotecas de suporte:

javascript

Copy code

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import { Provider } from 'react-redux';

import store from './store';

import { loadUser } from './actions/authActions';

import setAuthToken from './utils/setAuthToken';

import PrivateRoute from './components/routing/PrivateRoute';

import Navbar from './components/layout/Navbar';

import Landing from './components/layout/Landing';

import Login from './components/auth/Login';

import Register from './components/auth/Register';

import Dashboard from './components/dashboard/Dashboard';

import CreateProfile from './components/profile-forms/CreateProfile';

import EditProfile from './components/profile-forms/EditProfile';

import AddExperience from './components/profile-forms/AddExperience';

import AddEducation from './components/profile-forms/AddEducation';

import Profiles from './components/profiles/Profiles';

import Profile from './components/profile/Profile';

import Posts from './components/posts/Posts';

import Post from './components/post/Post';

// Verifica se há um token de autenticação e configura o header

if (localStorage.token) {

setAuthToken(localStorage.token);

}

function App() {

React.useEffect(() => {

store.dispatch(loadUser());

}, []);

return (

<Provider store={store}>

<Router>

<Navbar />

<Route exact path="/" component={Landing} />

<Switch>

<Route exact path="/register" component={Register} />

<Route exact path="/login" component={Login} />

<PrivateRoute exact path="/dashboard" component={Dashboard} />

<PrivateRoute

exact

path="/create-profile"

component={CreateProfile}

/>

<PrivateRoute

exact

path="/edit-profile"

component={EditProfile}

/>

<PrivateRoute

exact

path="/add-experience"

component={AddExperience}

/>

<PrivateRoute

exact

path="/add-education"

component={AddEducation}

/>

<Route exact path="/profiles" component={Profiles} />

<Route exact path="/profile/:id" component={Profile} />

<PrivateRoute exact path="/posts" component={Posts} />

<PrivateRoute exact path="/posts/:id" component={Post} />

</Switch>

</Router>

</Provider>

);

}

export default App;

Neste exemplo, o código usa o framework React para criar uma aplicação SPA que inclui autenticação de usuários, roteamento, gerenciamento de estado e componentes para renderizar a interface do usuário. Ele usa o Redux para gerenciar o estado da aplicação, o React Router para gerenciar as rotas e vários componentes personalizados para renderizar diferentes partes da interface do usuário. Além disso, o código inclui ações e utilitários personalizados para gerenciar a autenticação e a configuração do header da solicitação HTTP. Tudo isso pode parecer um pouco complexo, mas é típico para uma aplicação SPA moderna que usa o React e outras bibliotecas populares de suporte.

Rogério Soares (Merovingio)

quais comando preciso executar no terminal para a aplicação anterior?

Para executar a aplicação SPA apresentada no exemplo anterior, você precisaria seguir estes passos:

Certifique-se de que você tenha o Node.js e o gerenciador de pacotes npm instalados em seu computador.

Baixe o código do exemplo em um diretório local e navegue até o diretório raiz do projeto usando o terminal.

Execute o comando npm install para instalar todas as dependências do projeto especificadas no arquivo package.json.

Execute o comando npm start para iniciar o servidor de desenvolvimento. Isso compilará o código, iniciará um servidor local e abrirá a aplicação em um navegador da web.

Você pode então visualizar e interagir com a aplicação SPA no navegador.

Este é um processo geral, mas os comandos específicos podem variar dependendo de como o projeto está configurado e das dependências que ele usa. Em alguns casos, pode ser necessário configurar um banco de dados ou executar outras tarefas de configuração antes de iniciar a aplicação.

Rogério Soares (Merovingio)

liste todos os comandos por favor e o conteúdo do package.json

Os comandos que você precisaria executar no terminal para executar a aplicação SPA seriam:

npm install: instala todas as dependências do projeto especificadas no arquivo package.json.

npm start: inicia o servidor de desenvolvimento e abre a aplicação em um navegador da web.

npm test: executa os testes automatizados.

npm run build: cria uma versão otimizada para produção do aplicativo.

O conteúdo do arquivo package.json pode variar dependendo do projeto, mas um exemplo para o código da aplicação SPA apresentado anteriormente pode ser:

json

Copy code

{

"name": "my-app",

"version": "0.1.0",

"private": true,

"dependencies": {

"react": "^17.0.2",

"react-dom": "^17.0.2",

"react-redux": "^7.2.5",

"react-router-dom": "^5.3.0",

"redux": "^4.1.2",

"redux-devtools-extension": "^2.13.9",

"redux-thunk": "^2.4.1"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test",

"eject": "react-scripts eject"

}

}

Nesse exemplo, as dependências incluem o React, o React Router, o Redux e outras bibliotecas populares de suporte. Os scripts incluem os comandos necessários para executar a aplicação e os testes automatizados. Note que a maioria das dependências incluídas neste exemplo são para o desenvolvimento da aplicação e não são necessárias em produção.

Cite outro exemplo bem completo com react native

Aqui está um exemplo de código React Native que implementa uma lista de tarefas (To-Do List) simples:

javascript

Copy code

import React, { useState } from 'react';

import {

SafeAreaView,

StyleSheet,

TextInput,

TouchableOpacity,

FlatList,

Text,

View,

} from 'react-native';

export default function App() {

const [inputValue, setInputValue] = useState('');

const [tasks, setTasks] = useState([]);

const handlePress = () => {

if (inputValue) {

setTasks([...tasks, inputValue]);

setInputValue('');

}

};

const handleDelete = (index) => {

const newTasks = [...tasks];

newTasks.splice(index, 1);

setTasks(newTasks);

};

const renderItem = ({ item, index }) => (

<View style={styles.taskContainer}>

<Text style={styles.taskText}>{item}</Text>

<TouchableOpacity onPress={() => handleDelete(index)}>

<Text style={styles.deleteText}>Delete</Text>

</TouchableOpacity>

</View>

);

return (

<SafeAreaView style={styles.container}>

<View style={styles.inputContainer}>

<TextInput

style={styles.input}

value={inputValue}

onChangeText={setInputValue}

placeholder="Add a task..."

/>

<TouchableOpacity style={styles.button} onPress={handlePress}>

<Text style={styles.buttonText}>Add</Text>

</TouchableOpacity>

</View>

<FlatList

data={tasks}

renderItem={renderItem}

keyExtractor={(item, index) => index.toString()}

/>

</SafeAreaView>

);

}

const styles = StyleSheet.create({

container: {

flex: 1,

backgroundColor: '#fff',

},

inputContainer: {

flexDirection: 'row',

alignItems: 'center',

padding: 16,

borderBottomWidth: 1,

borderBottomColor: '#ccc',

},

input: {

flex: 1,

height: 40,

marginRight: 8,

paddingHorizontal: 8,

borderWidth: 1,

borderColor: '#ccc',

borderRadius: 4,

},

button: {

paddingHorizontal: 16,

paddingVertical: 8,

backgroundColor: '#2196f3',

borderRadius: 4,

},

buttonText: {

color: '#fff',

fontWeight: 'bold',

},

taskContainer: {

flexDirection: 'row',

alignItems: 'center',

padding: 16,

borderBottomWidth: 1,

borderBottomColor: '#ccc',

},

taskText: {

flex: 1,

fontWeight: 'bold',

},

deleteText: {

color: '#f44336',

fontWeight: 'bold',

},

});

Este exemplo utiliza os componentes de interface de usuário do React Native, como o SafeAreaView, StyleSheet, TextInput, TouchableOpacity, FlatList e Text para criar uma lista de tarefas. A lista é exibida em um FlatList, que é um componente otimizado para exibir grandes quantidades de dados roláveis. A entrada do usuário é gerenciada por meio do estado do React usando o useState hook. As tarefas são armazenadas em um estado de matriz (tasks) e podem ser adicionadas ou removidas por meio de funções que atualizam esse estado. O código também utiliza o StyleSheet.create para definir estilos para os componentes, que são aplicados automaticamente aos componentes no momento da renderização.

Faça um aplicativo react native que com o celular exiba um pequeno mapa e as coordenadas latitude e longitude e um botão para envio dela por alguma rede social

Segue abaixo um exemplo de código para um aplicativo React Native que exibe um mapa com as coordenadas de latitude e longitude, e tem um botão para compartilhar as coordenadas por meio de alguma rede social:

javascript

Copy code

import React, { useState, useEffect } from 'react';

import { StyleSheet, Text, View, TouchableOpacity } from 'react-native';

import MapView, { Marker } from 'react-native-maps';

import \* as Location from 'expo-location';

import \* as Sharing from 'expo-sharing';

export default function App() {

const [location, setLocation] = useState(null);

useEffect(() => {

(async () => {

let { status } = await Location.requestForegroundPermissionsAsync();

if (status !== 'granted') {

alert('Permission to access location was denied');

return;

}

let { coords } = await Location.getCurrentPositionAsync({});

setLocation(coords);

})();

}, []);

const handleShare = async () => {

if (!location) {

return;

}

const { latitude, longitude } = location;

const message = `Latitude: ${latitude}, Longitude: ${longitude}`;

await Sharing.shareAsync(message);

};

return (

<View style={styles.container}>

{location && (

<MapView style={styles.map} initialRegion={{

latitude: location.latitude,

longitude: location.longitude,

latitudeDelta: 0.0922,

longitudeDelta: 0.0421,

}}>

<Marker coordinate={{ latitude: location.latitude, longitude: location.longitude }} />

</MapView>

)}

{location && (

<View style={styles.coordinatesContainer}>

<Text style={styles.coordinatesText}>Latitude: {location.latitude}</Text>

<Text style={styles.coordinatesText}>Longitude: {location.longitude}</Text>

<TouchableOpacity style={styles.shareButton} onPress={handleShare}>

<Text style={styles.shareButtonText}>Share</Text>

</TouchableOpacity>

</View>

)}

</View>

);

}

const styles = StyleSheet.create({

container: {

flex: 1,

},

map: {

flex: 1,

},

coordinatesContainer: {

position: 'absolute',

bottom: 0,

left: 0,

right: 0,

backgroundColor: 'white',

padding: 16,

borderTopWidth: 1,

borderTopColor: '#ccc',

flexDirection: 'row',

alignItems: 'center',

},

coordinatesText: {

flex: 1,

fontWeight: 'bold',

},

shareButton: {

backgroundColor: '#2196f3',

borderRadius: 4,

paddingHorizontal: 16,

paddingVertical: 8,

},

shareButtonText: {

color: 'white',

fontWeight: 'bold',

},

});

Este aplicativo usa o pacote react-native-maps para exibir o mapa e o pacote expo-location para obter as coordenadas de latitude e longitude do dispositivo. Além disso, usa o pacote expo-sharing para compartilhar as coordenadas por meio de alguma rede social.

Para que o aplicativo funcione corretamente, é necessário configurar o app.json para incluir a chave de API do Google Maps. Para fazer isso, crie um arquivo app.json na raiz do projeto com o seguinte conteúdo:

json

Copy code

{

"expo": {

"android": {

"package": "com.example.app",

"versionCode": 1,

"config": {

"googleMaps": {

"apiKey": "YOUR

Rogério Soares (Merovingio)

continue o código

import React, { useState, useEffect } from 'react';

import { View, Text, StyleSheet, TouchableOpacity } from 'react-native';

import MapView, { Marker } from 'react-native-maps';

import \* as Location from 'expo-location';

export default function App() {

const [location, setLocation] = useState(null);

useEffect(() => {

(async () => {

let { status } = await Location.requestForegroundPermissionsAsync();

if (status !== 'granted') {

console.log('Permission to access location was denied');

return;

}

scss

Copy code

let currentLocation = await Location.getCurrentPositionAsync({});

setLocation(currentLocation);

})();

}, []);

const handleShareLocation = () => {

const { latitude, longitude } = location.coords;

const message = My current location: ${latitude}, ${longitude};

arduino

Copy code

// Send message through social network here

};

return (

<View style={styles.container}>

{location ? (

<>

<MapView

style={styles.map}

region={{

latitude: location.coords.latitude,

longitude: location.coords.longitude,

latitudeDelta: 0.01,

longitudeDelta: 0.01,

}}

>

<Marker

coordinate={{

latitude: location.coords.latitude,

longitude: location.coords.longitude,

}}

/>

</MapView>

<View style={styles.infoContainer}>

<Text style={styles.infoText}>

Latitude: {location.coords.latitude}

</Text>

<Text style={styles.infoText}>

Longitude: {location.coords.longitude}

</Text>

<TouchableOpacity

style={styles.shareButton}

onPress={handleShareLocation}

>

<Text style={styles.shareButtonText}>Share on Social Network</Text>

</TouchableOpacity>

</View>

</>

) : (

<Text style={styles.loadingText}>Loading...</Text>

)}

</View>

);

}

const styles = StyleSheet.create({

container: {

flex: 1,

},

map: {

flex: 1,

},

infoContainer: {

position: 'absolute',

bottom: 0,

left: 0,

right: 0,

backgroundColor: '#fff',

padding: 16,

borderTopWidth: 1,

borderTopColor: '#ccc',

flexDirection: 'row',

alignItems: 'center',

},

infoText: {

marginRight: 16,

},

shareButton: {

paddingHorizontal: 16,

paddingVertical: 8,

backgroundColor: '#2196f3',

borderRadius: 4,

},

shareButtonText: {

color: '#fff',

fontWeight: 'bold',

},

loadingText: {

flex: 1,

textAlign: 'center',

textAlignVertical: 'center',

fontSize: 24,

},

});

// OBS: Lembre-se de instalar as dependências necessárias, como o 'expo-location' e o 'react-native-maps'.

Rogério Soares (Merovingio)

termine esse código: {

"expo": {

"android": {

"package": "com.example.app",

"versionCode": 1,

"config": {

"googleMaps": {

"apiKey": "YOUR

API\_KEY\_HERE"

}

}

},

"ios": {

"bundleIdentifier": "com.example.app",

"buildNumber": "1.0.0",

"config": {

"googleMaps": {

"apiKey": "YOUR\_API\_KEY\_HERE"

}

}

},

"name": "My Awesome App",

"slug": "my-awesome-app",

"version": "1.0.0",

"sdkVersion": "41.0.0",

"platforms": ["ios", "android"],

"description": "An awesome app built with React Native and Expo!",

"icon": "./assets/icon.png",

"splash": {

"image": "./assets/splash.png",

"resizeMode": "contain",

"backgroundColor": "#ffffff"

},

"updates": {

"fallbackToCacheTimeout": 0

},

"assetBundlePatterns": ["\*\*/\*"],

"iosStatusBar": {

"barStyle": "dark-content",

"backgroundColor": "#ffffff"

},

"androidStatusBar": {

"barStyle": "dark-content",

"backgroundColor": "#ffffff"

}

}

}

# Testes unitários e testes de integração em JavaScript

Testes unitários e testes de integração em JavaScript

Testes unitários e testes de integração são técnicas de testes de software utilizadas para garantir que o código esteja funcionando corretamente. No JavaScript, existem várias ferramentas e bibliotecas que podem ser utilizadas para realizar esses tipos de testes.

Os testes unitários são utilizados para testar pequenas partes do código, como funções ou métodos específicos. Eles são escritos para garantir que a lógica subjacente esteja funcionando corretamente e que os resultados esperados sejam produzidos. Para escrever testes unitários em JavaScript, pode-se usar frameworks como Mocha, Jasmine ou Jest. Esses frameworks fornecem uma estrutura para definir e executar testes, bem como recursos para verificar se os resultados estão corretos.

Por outro lado, os testes de integração são usados para testar como as diferentes partes do sistema funcionam juntas. Eles geralmente envolvem testar o comportamento de várias funções ou módulos interconectados e verificar se eles estão integrados corretamente. Para realizar testes de integração em JavaScript, pode-se usar ferramentas como Selenium ou Cypress. Essas ferramentas permitem que os testes sejam escritos como scripts que simulam a interação do usuário com a aplicação.

É importante lembrar que os testes unitários e os testes de integração devem ser complementares e não substituem uns aos outros. Os testes unitários garantem que as partes individuais do código estejam funcionando corretamente, enquanto os testes de integração garantem que elas funcionem corretamente em conjunto com outras partes do sistema. Juntos, eles ajudam a garantir que o código seja robusto e livre de erros.

Rogério Soares (Merovingio)

Cite vários exemplos de testes unitários em javascript e ecma script 2021

A seguir, alguns exemplos de testes unitários que podem ser escritos em JavaScript e ECMAScript 2021:

Teste de uma função que soma dois números:

javascript

Copy code

function soma(a, b) {

return a + b;

}

describe("Teste da função soma", () => {

it("deve retornar 4 quando passados 2 e 2 como parâmetros", () => {

expect(soma(2, 2)).toBe(4);

});

it("deve retornar 0 quando passados -2 e 2 como parâmetros", () => {

expect(soma(-2, 2)).toBe(0);

});

});

Teste de uma função que retorna o número de caracteres de uma string:

javascript

Copy code

function contarCaracteres(str) {

return str.length;

}

describe("Teste da função contarCaracteres", () => {

it("deve retornar 5 para a string 'hello'", () => {

expect(contarCaracteres("hello")).toBe(5);

});

it("deve retornar 0 para uma string vazia", () => {

expect(contarCaracteres("")).toBe(0);

});

});

Teste de uma classe que representa uma calculadora:

javascript

Copy code

class Calculadora {

somar(a, b) {

return a + b;

}

subtrair(a, b) {

return a - b;

}

}

describe("Teste da classe Calculadora", () => {

const calculadora = new Calculadora();

it("deve somar corretamente dois números", () => {

expect(calculadora.somar(2, 2)).toBe(4);

});

it("deve subtrair corretamente dois números", () => {

expect(calculadora.subtrair(5, 3)).toBe(2);

});

});

Teste de uma função assíncrona que faz uma requisição HTTP:

javascript

Copy code

async function buscarDados(url) {

const resposta = await fetch(url);

const dados = await resposta.json();

return dados;

}

describe("Teste da função buscarDados", () => {

it("deve retornar um objeto com o nome 'John' quando a URL 'https://jsonplaceholder.typicode.com/users/1' for passada como parâmetro", async () => {

const dados = await buscarDados("https://jsonplaceholder.typicode.com/users/1");

expect(dados.name).toBe("John");

});

});

Esses são apenas alguns exemplos de testes unitários que podem ser escritos em JavaScript e ECMAScript 2021. A escolha dos testes dependerá do tipo de código que está sendo testado e dos requisitos de qualidade do projeto.

Rogério Soares (Merovingio)

monte alguns exemplos de teste unitários com Mocha, passo a passo, com JavaScript e ECMAScript 2021

Aqui estão alguns exemplos passo a passo de como escrever testes unitários com Mocha em JavaScript e ECMAScript 2021:

Teste de uma função que soma dois números:

javascript

Copy code

function soma(a, b) {

return a + b;

}

// Importa o Mocha e a biblioteca de assertividade Chai

const assert = require("chai").assert;

const mocha = require("mocha");

// Descreve o teste usando a função describe do Mocha

describe("Teste da função soma", () => {

// Define cada caso de teste usando a função it do Mocha

it("deve retornar 4 quando passados 2 e 2 como parâmetros", () => {

// Usa as funções de assertividade do Chai para testar o resultado da função

assert.equal(soma(2, 2), 4);

});

it("deve retornar 0 quando passados -2 e 2 como parâmetros", () => {

assert.equal(soma(-2, 2), 0);

});

});

Teste de uma função que retorna o número de caracteres de uma string:

javascript

Copy code

function contarCaracteres(str) {

return str.length;

}

const assert = require("chai").assert;

const mocha = require("mocha");

describe("Teste da função contarCaracteres", () => {

it("deve retornar 5 para a string 'hello'", () => {

assert.equal(contarCaracteres("hello"), 5);

});

it("deve retornar 0 para uma string vazia", () => {

assert.equal(contarCaracteres(""), 0);

});

});

Teste de uma classe que representa uma calculadora:

javascript

Copy code

class Calculadora {

somar(a, b) {

return a + b;

}

subtrair(a, b) {

return a - b;

}

}

const assert = require("chai").assert;

const mocha = require("mocha");

describe("Teste da classe Calculadora", () => {

const calculadora = new Calculadora();

it("deve somar corretamente dois números", () => {

assert.equal(calculadora.somar(2, 2), 4);

});

it("deve subtrair corretamente dois números", () => {

assert.equal(calculadora.subtrair(5, 3), 2);

});

});

Teste de uma função assíncrona que faz uma requisição HTTP:

javascript

Copy code

async function buscarDados(url) {

const resposta = await fetch(url);

const dados = await resposta.json();

return dados;

}

const assert = require("chai").assert;

const mocha = require("mocha");

describe("Teste da função buscarDados", () => {

it("deve retornar um objeto com o nome 'John' quando a URL 'https://jsonplaceholder.typicode.com/users/1' for passada como parâmetro", async () => {

const dados = await buscarDados("https://jsonplaceholder.typicode.com/users/1");

assert.equal(dados.name, "John");

});

});

Esses são apenas alguns exemplos de como escrever testes unitários com Mocha em JavaScript e ECMAScript 2021. A biblioteca de assertividade Chai oferece diversas funções para testar diferentes tipos de dados e expressões, e o Mocha ofere

Rogério Soares (Merovingio)

monte alguns exemplos de teste unitários com Jasmine, passo a passo com JavaScript e ECMAScript 2021

Aqui estão alguns exemplos passo a passo de como escrever testes unitários com Jasmine em JavaScript e ECMAScript 2021:

Teste de uma função que soma dois números:

javascript

Copy code

function soma(a, b) {

return a + b;

}

// Importa o Jasmine

const jasmine = require("jasmine");

// Descreve o teste usando a função describe do Jasmine

describe("Teste da função soma", () => {

// Define cada caso de teste usando a função it do Jasmine

it("deve retornar 4 quando passados 2 e 2 como parâmetros", () => {

// Usa a função expect do Jasmine para testar o resultado da função

expect(soma(2, 2)).toEqual(4);

});

it("deve retornar 0 quando passados -2 e 2 como parâmetros", () => {

expect(soma(-2, 2)).toEqual(0);

});

});

Teste de uma função que retorna o número de caracteres de uma string:

javascript

Copy code

function contarCaracteres(str) {

return str.length;

}

const jasmine = require("jasmine");

describe("Teste da função contarCaracteres", () => {

it("deve retornar 5 para a string 'hello'", () => {

expect(contarCaracteres("hello")).toEqual(5);

});

it("deve retornar 0 para uma string vazia", () => {

expect(contarCaracteres("")).toEqual(0);

});

});

Teste de uma classe que representa uma calculadora:

javascript

Copy code

class Calculadora {

somar(a, b) {

return a + b;

}

subtrair(a, b) {

return a - b;

}

}

const jasmine = require("jasmine");

describe("Teste da classe Calculadora", () => {

const calculadora = new Calculadora();

it("deve somar corretamente dois números", () => {

expect(calculadora.somar(2, 2)).toEqual(4);

});

it("deve subtrair corretamente dois números", () => {

expect(calculadora.subtrair(5, 3)).toEqual(2);

});

});

Teste de uma função assíncrona que faz uma requisição HTTP:

javascript

Copy code

async function buscarDados(url) {

const resposta = await fetch(url);

const dados = await resposta.json();

return dados;

}

const jasmine = require("jasmine");

describe("Teste da função buscarDados", () => {

it("deve retornar um objeto com o nome 'John' quando a URL 'https://jsonplaceholder.typicode.com/users/1' for passada como parâmetro", async () => {

const dados = await buscarDados("https://jsonplaceholder.typicode.com/users/1");

expect(dados.name).toEqual("John");

});

});

Esses são apenas alguns exemplos de como escrever testes unitários com Jasmine em JavaScript e ECMAScript 2021. O Jasmine oferece diversas funções para testar diferentes tipos de dados e expressões, como toEqual, toBe, toBeTruthy, entre outras. É possível também utilizar funções como beforeEach e afterEach para executar ações antes e depois de cada teste.

Rogério Soares (Merovingio)

monte alguns exemplos de teste unitários com Jest, passo a passo com JavaScript e ECMAScript 2021

Aqui estão alguns exemplos passo a passo de como escrever testes unitários com Jest em JavaScript e ECMAScript 2021:

Teste de uma função que soma dois números:

javascript

Copy code

function soma(a, b) {

return a + b;

}

describe("Teste da função soma", () => {

it("deve retornar 4 quando passados 2 e 2 como parâmetros", () => {

expect(soma(2, 2)).toEqual(4);

});

it("deve retornar 0 quando passados -2 e 2 como parâmetros", () => {

expect(soma(-2, 2)).toEqual(0);

});

});

Teste de uma função que retorna o número de caracteres de uma string:

javascript

Copy code

function contarCaracteres(str) {

return str.length;

}

describe("Teste da função contarCaracteres", () => {

it("deve retornar 5 para a string 'hello'", () => {

expect(contarCaracteres("hello")).toEqual(5);

});

it("deve retornar 0 para uma string vazia", () => {

expect(contarCaracteres("")).toEqual(0);

});

});

Teste de uma classe que representa uma calculadora:

javascript

Copy code

class Calculadora {

somar(a, b) {

return a + b;

}

subtrair(a, b) {

return a - b;

}

}

describe("Teste da classe Calculadora", () => {

const calculadora = new Calculadora();

it("deve somar corretamente dois números", () => {

expect(calculadora.somar(2, 2)).toEqual(4);

});

it("deve subtrair corretamente dois números", () => {

expect(calculadora.subtrair(5, 3)).toEqual(2);

});

});

Teste de uma função assíncrona que faz uma requisição HTTP:

javascript

Copy code

async function buscarDados(url) {

const resposta = await fetch(url);

const dados = await resposta.json();

return dados;

}

describe("Teste da função buscarDados", () => {

it("deve retornar um objeto com o nome 'John' quando a URL 'https://jsonplaceholder.typicode.com/users/1' for passada como parâmetro", async () => {

const dados = await buscarDados("https://jsonplaceholder.typicode.com/users/1");

expect(dados.name).toEqual("John");

});

});

O Jest também oferece diversas funções para testar diferentes tipos de dados e expressões, como toEqual, toBe, toBeTruthy, entre outras. Além disso, é possível utilizar funções como beforeEach e afterEach para executar ações antes e depois de cada teste, e describe para agrupar os testes em categorias. O Jest também possui recursos para testes de snapshot e mocks.

Rogério Soares (Merovingio)

mostre testes de integração em JavaScript, com as ferramentas como Selenium ou Cypress

passo a passo com JavaScript e ECMAScript 2021

Aqui estão alguns exemplos passo a passo de como escrever testes de integração em JavaScript com as ferramentas Selenium e Cypress em ECMAScript 2021:

Teste de integração com o Selenium

Para este exemplo, vamos criar um teste de integração que acessa uma página web, preenche um formulário e verifica se a página de confirmação é exibida corretamente.

Passo 1: Configurar o ambiente de teste

javascript

Copy code

const { Builder, By, Key, until } = require('selenium-webdriver');

describe('Teste de integração com Selenium', () => {

let driver;

before(async () => {

driver = await new Builder().forBrowser('chrome').build();

});

after(async () => {

await driver.quit();

});

Passo 2: Escrever o teste

javascript

Copy code

it('deve preencher e enviar o formulário corretamente', async () => {

await driver.get('https://exemplo.com/formulario');

await driver.findElement(By.name('nome')).sendKeys('Fulano');

await driver.findElement(By.name('email')).sendKeys('fulano@exemplo.com');

await driver.findElement(By.name('mensagem')).sendKeys('Isso é um teste de integração');

await driver.findElement(By.css('button[type=submit]')).sendKeys(Key.ENTER);

await driver.wait(until.urlContains('https://exemplo.com/confirmacao'), 10000);

const titulo = await driver.getTitle();

expect(titulo).toEqual('Página de Confirmação');

});

Este teste abre o navegador Google Chrome, acessa a página com o formulário, preenche os campos do formulário e envia. Em seguida, aguarda até que a página de confirmação seja carregada e verifica se o título da página é 'Página de Confirmação'.

Teste de integração com o Cypress

Para este exemplo, vamos criar um teste de integração que acessa uma página web, preenche um formulário e verifica se a página de confirmação é exibida corretamente.

Passo 1: Instalar o Cypress

css

Copy code

npm install cypress --save-dev

Passo 2: Configurar o ambiente de teste

javascript

Copy code

describe('Teste de integração com Cypress', () => {

beforeEach(() => {

cy.visit('https://exemplo.com/formulario');

});

Passo 3: Escrever o teste

javascript

Copy code

it('deve preencher e enviar o formulário corretamente', () => {

cy.get('input[name="nome"]').type('Fulano');

cy.get('input[name="email"]').type('fulano@exemplo.com');

cy.get('textarea[name="mensagem"]').type('Isso é um teste de integração');

cy.get('button[type="submit"]').click();

cy.url().should('include', 'https://exemplo.com/confirmacao');

cy.title().should('equal', 'Página de Confirmação');

});

});

Este teste acessa a página com o formulário, preenche os campos do formulário e envia. Em seguida, verifica se a URL da página é 'https://exemplo.com/confirmacao' e se o título da página é 'Página de Confirmação'.

Rogério Soares (Merovingio)

mostre testes de integração em JavaScript puro, se for possível

Sim, é possível escrever testes de integração em JavaScript puro sem o uso de ferramentas específicas como o Selenium ou Cypress. Aqui está um exemplo de como fazer isso:

Teste de integração com JavaScript puro

Para este exemplo, vamos criar um teste de integração que acessa uma API REST, envia uma solicitação POST com um objeto JSON e verifica se a resposta está correta.

Passo 1: Escrever o teste

javascript

Copy code

describe('Teste de integração com JavaScript puro', () => {

it('deve enviar uma solicitação POST para a API REST e receber a resposta correta', (done) => {

const xhr = new XMLHttpRequest();

xhr.open('POST', 'https://exemplo.com/api');

xhr.setRequestHeader('Content-Type', 'application/json');

xhr.onload = function() {

expect(xhr.status).toBe(200);

const response = JSON.parse(xhr.responseText);

expect(response.status).toBe('OK');

expect(response.message).toBe('Solicitação enviada com sucesso');

done();

};

xhr.onerror = function() {

done.fail('Erro ao enviar solicitação');

};

const data = {

nome: 'Fulano',

email: 'fulano@exemplo.com',

mensagem: 'Isso é um teste de integração'

};

xhr.send(JSON.stringify(data));

});

});

Este teste envia uma solicitação POST para a API REST em https://exemplo.com/api com um objeto JSON contendo as informações do formulário. Em seguida, verifica se a resposta HTTP é 200 e se a resposta JSON contém os valores esperados para as chaves 'status' e 'message'. O uso da função 'done()' é necessário para indicar que o teste foi concluído. Se ocorrer um erro ao enviar a solicitação, a função 'done.fail()' é chamada para indicar que o teste falhou.

Lembre-se de que testes de integração podem ser mais lentos e frágeis do que testes unitários. Portanto, é importante escrevê-los com cuidado e garantir que o ambiente de teste seja configurado corretamente. Além disso, é recomendável separar os testes de integração dos testes unitários e executá-los em um ambiente de teste separado.

Rogério Soares (Merovingio)

mostre testes unitários em JavaScript puro, se for possível

Sim, é possível escrever testes unitários em JavaScript puro sem o uso de frameworks ou bibliotecas. Aqui está um exemplo de como fazer isso:

Teste unitário com JavaScript puro

Para este exemplo, vamos criar um teste unitário simples que verifica se a função 'soma' retorna o resultado correto para dois valores passados como parâmetro.

Passo 1: Escrever a função 'soma'

javascript

Copy code

function soma(a, b) {

return a + b;

}

Esta função recebe dois parâmetros numéricos e retorna a soma deles.

Passo 2: Escrever o teste

javascript

Copy code

function testSoma() {

const resultado = soma(2, 3);

if (resultado !== 5) {

throw new Error('Erro no teste: soma(2, 3) deve ser igual a 5, mas é igual a ' + resultado);

}

}

testSoma();

Este teste chama a função 'soma' com os valores 2 e 3 e verifica se o resultado é igual a 5. Se o resultado for diferente, uma exceção será lançada com uma mensagem de erro indicando que o teste falhou.

Lembre-se de que este é apenas um exemplo simples e que em projetos reais é recomendável o uso de bibliotecas ou frameworks de testes unitários para facilitar a escrita e execução de testes. Além disso, é importante escrever testes que cubram todos os casos possíveis e que sejam fáceis de ler e entender.

# Uso de módulos em JavaScript (CommonJS, AMD, ES6)

Uso de módulos em JavaScript (CommonJS, AMD, ES2021)

Em JavaScript, os módulos são usados para organizar o código em unidades isoladas e reutilizáveis, o que pode ajudar a melhorar a manutenção e a escalabilidade de um projeto. Existem várias formas de se trabalhar com módulos em JavaScript, entre elas:

CommonJS: É um formato de módulo utilizado principalmente no ambiente Node.js. Nele, os módulos são carregados de forma síncrona e as dependências são tratadas de forma explícita através da função require. Para exportar um módulo, utiliza-se o objeto module.exports.

Exemplo de utilização:

css

Copy code

// Arquivo: modulo.js

function somar(a, b) {

return a + b;

}

module.exports = {

somar: somar

};

javascript

Copy code

// Arquivo: index.js

const modulo = require('./modulo');

console.log(modulo.somar(2, 3)); // 5

AMD (Asynchronous Module Definition): É um formato de módulo utilizado principalmente no ambiente do navegador, onde os módulos são carregados de forma assíncrona. Para definir um módulo, utiliza-se a função define, que permite especificar as dependências e uma função de retorno que define o comportamento do módulo.

Exemplo de utilização:

css

Copy code

// Arquivo: modulo.js

define(function() {

function somar(a, b) {

return a + b;

}

return {

somar: somar

};

});

javascript

Copy code

// Arquivo: index.js

require(['modulo'], function(modulo) {

console.log(modulo.somar(2, 3)); // 5

});

ES2021 (ESM - ECMAScript Modules): É uma forma de definir módulos nativamente em JavaScript, disponível a partir da versão ES6. Os módulos são carregados de forma assíncrona e as dependências são tratadas de forma explícita através da palavra-chave import. Para exportar um módulo, utiliza-se a palavra-chave export.

Exemplo de utilização:

javascript

Copy code

// Arquivo: modulo.js

function somar(a, b) {

return a + b;

}

export { somar };

javascript

Copy code

// Arquivo: index.js

import { somar } from './modulo.js';

console.log(somar(2, 3)); // 5

Cada formato de módulo possui suas particularidades e pode ser mais adequado para diferentes contextos e necessidades. É importante conhecer esses diferentes formatos e escolher aquele que melhor se adapta ao seu projeto e ambiente de execução.

Rogério Soares (Merovingio)

monte um script de calculo IMC, que mostre na página html o resultado usando sistema de módulos, arquivos separados javascript puro e ES 2021

Aqui está um exemplo de como implementar um script de cálculo de IMC usando sistema de módulos em JavaScript puro e ES2021:

JavaScript puro (CommonJS)

Crie um arquivo imc.js para definir a função de cálculo de IMC:

java

Copy code

function calcularIMC(peso, altura) {

const imc = peso / (altura \* altura);

return imc.toFixed(2);

}

module.exports = calcularIMC;

Crie um arquivo index.js para utilizar a função de cálculo de IMC e exibir o resultado na página HTML:

javascript

Copy code

const calcularIMC = require('./imc');

const formulario = document.querySelector('form');

const resultado = document.querySelector('#resultado');

formulario.addEventListener('submit', (event) => {

event.preventDefault();

const peso = parseFloat(formulario.querySelector('#peso').value);

const altura = parseFloat(formulario.querySelector('#altura').value);

const imc = calcularIMC(peso, altura);

resultado.textContent = `Seu IMC é ${imc}`;

});

Adicione o seguinte HTML à sua página:

php

Copy code

<form>

<label for="peso">Peso (kg):</label>

<input type="number" id="peso" required>

<br>

<label for="altura">Altura (m):</label>

<input type="number" id="altura" step="0.01" required>

<br>

<button type="submit">Calcular</button>

</form>

<div id="resultado"></div>

ES2021 (ESM - ECMAScript Modules)

Crie um arquivo imc.mjs para definir a função de cálculo de IMC:

javascript

Copy code

function calcularIMC(peso, altura) {

const imc = peso / (altura \* altura);

return imc.toFixed(2);

}

export default calcularIMC;

Crie um arquivo index.mjs para utilizar a função de cálculo de IMC e exibir o resultado na página HTML:

javascript

Copy code

import calcularIMC from './imc.js';

const formulario = document.querySelector('form');

const resultado = document.querySelector('#resultado');

formulario.addEventListener('submit', (event) => {

event.preventDefault();

const peso = parseFloat(formulario.querySelector('#peso').value);

const altura = parseFloat(formulario.querySelector('#altura').value);

const imc = calcularIMC(peso, altura);

resultado.textContent = `Seu IMC é ${imc}`;

});

Adicione o seguinte HTML à sua página:

php

Copy code

<form>

<label for="peso">Peso (kg):</label>

<input type="number" id="peso" required>

<br>

<label for="altura">Altura (m):</label>

<input type="number" id="altura" step="0.01" required>

<br>

<button type="submit">Calcular</button>

</form>

<div id="resultado"></div>

Lembre-se de que, para utilizar módulos ES2021 em navegadores, é necessário configurar o atributo type="module" no elemento <script> que importa o arquivo index.mjs. Por exemplo:

php

Copy code

<script type="module" src="index.mjs"></script>

Rogério Soares (Merovingio)

monte um script de calculo de parse de corrida em km/min e em km/h bem como as calorias queimadas dado o peso da pessoa, distância percorrida e tempo em h:m:s e distância em metros, que mostre na página html o resultado usando sistema de módulos, arquivos separados javascript puro e ES 2021

Aqui está um exemplo de como implementar um script de cálculo de parse de corrida em km/min e km/h e calorias queimadas, usando sistema de módulos em JavaScript puro e ES2021:

JavaScript puro (CommonJS)

Crie um arquivo calculos.js para definir as funções de cálculo de parse, calorias e conversão de tempo:

javascript

Copy code

function calcularParse(distancia, tempo) {

const segundos = tempoParaSegundos(tempo);

const minutos = segundos / 60;

const kmPorMinuto = distancia / minutos;

const kmPorHora = kmPorMinuto \* 60;

return {

kmPorMinuto: kmPorMinuto.toFixed(2),

kmPorHora: kmPorHora.toFixed(2),

};

}

function calcularCalorias(peso, distancia) {

const calorias = peso \* distancia \* 0.001036;

return calorias.toFixed(2);

}

function tempoParaSegundos(tempo) {

const [horas, minutos, segundos] = tempo.split(':');

return (horas \* 3600) + (minutos \* 60) + seconds;

}

module.exports = {

calcularParse,

calcularCalorias,

};

Crie um arquivo index.js para utilizar as funções de cálculo de parse, calorias e exibir os resultados na página HTML:

javascript

Copy code

const { calcularParse, calcularCalorias } = require('./calculos');

const formulario = document.querySelector('form');

const resultado = document.querySelector('#resultado');

formulario.addEventListener('submit', (event) => {

event.preventDefault();

const peso = parseFloat(formulario.querySelector('#peso').value);

const distancia = parseFloat(formulario.querySelector('#distancia').value);

const tempo = formulario.querySelector('#tempo').value;

const { kmPorMinuto, kmPorHora } = calcularParse(distancia, tempo);

const calorias = calcularCalorias(peso, distancia);

resultado.innerHTML = `

<p>Parse: ${kmPorMinuto} km/min ou ${kmPorHora} km/h</p>

<p>Calorias queimadas: ${calorias}</p>

`;

});

Adicione o seguinte HTML à sua página:

php

Copy code

<form>

<label for="peso">Peso (kg):</label>

<input type="number" id="peso" required>

<br>

<label for="distancia">Distância (m):</label>

<input type="number" id="distancia" required>

<br>

<label for="tempo">Tempo (h:m:s):</label>

<input type="text" id="tempo" required>

<br>

<button type="submit">Calcular</button>

</form>

<div id="resultado"></div>

ES2021 (ESM - ECMAScript Modules)

Crie um arquivo calculos.mjs para definir as funções de cálculo de parse, calorias e conversão de tempo:

javascript

Copy code

function calcularParse(distancia, tempo) {

const segundos = tempoParaSegundos(tempo);

const minutos = segundos / 60;

const kmPorMinuto = distancia / minutos;

const kmPorHora = kmPorMinuto \* 60;

return {

kmPorMinuto: kmPorMinuto.toFixed(2),

kmPorHora: kmPorHora.toFixed(2),

};

}

function calcularCalorias(peso, distancia) {

const

Rogério Soares (Merovingio)

continue

calorias = peso \* distancia \* 0.001036;

return calorias.toFixed(2);

}

function tempoParaSegundos(tempo) {

const [horas, minutos, segundos] = tempo.split(':');

return (horas \* 3600) + (minutos \* 60) + seconds;

}

export {

calcularParse,

calcularCalorias,

};

kotlin

Copy code

2. Crie um arquivo `index.mjs` para utilizar as funções de cálculo de parse, calorias e exibir os resultados na página HTML:

import { calcularParse, calcularCalorias } from './calculos.mjs';

const formulario = document.querySelector('form');

const resultado = document.querySelector('#resultado');

formulario.addEventListener('submit', (event) => {

event.preventDefault();

const peso = parseFloat(formulario.querySelector('#peso').value);

const distancia = parseFloat(formulario.querySelector('#distancia').value);

const tempo = formulario.querySelector('#tempo').value;

const { kmPorMinuto, kmPorHora } = calcularParse(distancia, tempo);

const calorias = calcularCalorias(peso, distancia);

resultado.innerHTML = <p>Parse: ${kmPorMinuto} km/min ou ${kmPorHora} km/h</p> <p>Calorias queimadas: ${calorias}</p> ;

});

css

Copy code

3. Adicione o seguinte HTML à sua página:

<form>

<label for="peso">Peso (kg):</label>

<input type="number" id="peso" required>

<br>

<label for="distancia">Distância (m):</label>

<input type="number" id="distancia" required>

<br>

<label for="tempo">Tempo (h:m:s):</label>

<input type="text" id="tempo" required>

<br>

<button type="submit">Calcular</button>

</form>

<div id="resultado"></div>

```

Com esses exemplos, é possível ter uma ideia de como utilizar módulos em JavaScript para implementar um script de cálculo de parse, calorias e conversão de tempo. Além disso, é importante lembrar que a escolha entre CommonJS e ES Modules depende do ambiente em que o script será executado, e é importante garantir que o ambiente seja compatível com o módulo escolhido.

Rogério Soares (Merovingio)

Monte um jogo da velha. Use SOLID e módulos para separar muito bem cada script e monte a página html estilizada com css3

Vou criar um exemplo básico de um jogo da velha utilizando SOLID e módulos em JavaScript. Para isso, vou seguir os seguintes passos:

Criar um módulo para representar cada conceito do jogo (tabuleiro, jogador, partida) e que possam ser usados de forma independente em outras partes do código. Para isso, vou criar as seguintes classes:

Tabuleiro: representa o tabuleiro do jogo, com suas casas e os métodos para marcar uma casa com o símbolo do jogador atual, verificar se há um vencedor e reiniciar o jogo.

Jogador: representa um jogador do jogo, com suas informações básicas (nome, símbolo).

Partida: representa uma partida do jogo da velha, com dois jogadores e o tabuleiro. Controla o turno dos jogadores, verifica o fim da partida e anuncia o vencedor.

Criar um módulo para representar a interação com a interface gráfica (HTML e CSS). Para isso, vou criar a classe JogoDaVelhaUI, que será responsável por criar os elementos HTML do jogo (tabuleiro, botões de reinício e informações de vitória) e gerenciar os eventos de clique nos botões e atualização da interface.

Juntar tudo em um arquivo index.js que instancie os objetos necessários e faça a inicialização do jogo.

Adicionar um arquivo style.css para estilizar o jogo da velha.

Com isso em mente, vamos ao código:

tabuleiro.js

javascript

Copy code

class Tabuleiro {

constructor() {

this.\_casas = Array(9).fill(null);

}

marcarCasa(casa, jogador) {

if (this.\_casas[casa] !== null) {

throw new Error('Casa já marcada');

}

this.\_casas[casa] = jogador.simbolo;

}

verificarVencedor() {

const posicoesVencedoras = [

// Linhas

[0, 1, 2],

[3, 4, 5],

[6, 7, 8],

// Colunas

[0, 3, 6],

[1, 4, 7],

[2, 5, 8],

// Diagonais

[0, 4, 8],

[2, 4, 6],

];

for (const [a, b, c] of posicoesVencedoras) {

if (

this.\_casas[a] !== null &&

this.\_casas[a] === this.\_casas[b] &&

this.\_casas[a] === this.\_casas[c]

) {

return this.\_casas[a];

}

}

if (this.\_casas.every((casa) => casa !== null)) {

return 'EMPATE';

}

return null;

}

reiniciar() {

this.\_casas.fill(null);

}

getCasas() {

return this.\_casas.slice();

}

}

export default Tabuleiro;

jogador.js

javascript

Copy code

class Jogador {

constructor(nome, simbolo) {

this.nome = nome;

this.simbolo = simbolo;

}

}

export default Jogador;

partida.js

javascript

Copy code

import Tabuleiro from './tabuleiro.js

Rogério Soares (Merovingio)

continue o código

javascript

Copy code

import Jogador from './jogador.js';

class Partida {

constructor(jogador1, jogador2) {

this.\_tabuleiro = new Tabuleiro();

this.\_jogadores = [jogador1, jogador2];

this.\_vezDoJogador = 0;

}

jogar(casa) {

const jogadorAtual = this.\_jogadores[this.\_vezDoJogador];

this.\_tabuleiro.marcarCasa(casa, jogadorAtual);

const vencedor = this.\_tabuleiro.verificarVencedor();

if (vencedor !== null) {

return vencedor;

}

this.\_trocarJogador();

return null;

}

\_trocarJogador() {

this.\_vezDoJogador = (this.\_vezDoJogador + 1) % 2;

}

reiniciar() {

this.\_tabuleiro.reiniciar();

this.\_vezDoJogador = 0;

}

getTabuleiro() {

return this.\_tabuleiro.getCasas();

}

getJogadorAtual() {

return this.\_jogadores[this.\_vezDoJogador];

}

}

export default Partida;

jogo-da-velha-ui.js

javascript

Copy code

class JogoDaVelhaUI {

constructor(container) {

this.\_container = container;

this.\_partida = null;

this.\_botoes = [];

this.\_mensagemVitoria = null;

this.\_botaoReiniciar = null;

}

iniciar() {

this.\_criarTabuleiro();

this.\_partida = new Partida(

new Jogador('Jogador 1', 'X'),

new Jogador('Jogador 2', 'O')

);

this.\_atualizarInterface();

}

\_criarTabuleiro() {

const tabuleiro = document.createElement('div');

tabuleiro.classList.add('tabuleiro');

for (let i = 0; i < 9; i++) {

const botao = document.createElement('button');

botao.classList.add('casa');

botao.addEventListener('click', () => this.\_jogar(i));

tabuleiro.appendChild(botao);

this.\_botoes.push(botao);

}

this.\_container.appendChild(tabuleiro);

this.\_botaoReiniciar = document.createElement('button');

this.\_botaoReiniciar.classList.add('botao-reiniciar');

this.\_botaoReiniciar.addEventListener('click', () => this.\_reiniciar());

this.\_container.appendChild(this.\_botaoReiniciar);

this.\_mensagemVitoria = document.createElement('div');

this.\_mensagemVitoria.classList.add('mensagem-vitoria');

this.\_container.appendChild(this.\_mensagemVitoria);

}

\_jogar(casa) {

const vencedor = this.\_partida.jogar(casa);

this.\_atualizarInterface();

if (vencedor !== null) {

this.\_mostrarVencedor(vencedor);

}

}

\_reiniciar() {

this.\_partida.reiniciar();

this.\_atualizarInterface();

this.\_esconderVencedor();

}

\_atualizarInterface() {

const casas = this.\_partida.getTabuleiro();

for (let i = 0; i < casas.length; i++)

Rogério Soares (Merovingio)

continue o código

javascript

Copy code

const botao = this.\_botoes[i];

const valor = casas[i];

botao.textContent = valor !== null ? valor : '';

}

const jogadorAtual = this.\_partida.getJogadorAtual();

this.\_botaoReiniciar.textContent = `Reiniciar (${jogadorAtual.getSimbolo()})`;

}

\_mostrarVencedor(vencedor) {

const mensagem = `${vencedor.getNome()} venceu!`;

this.\_mensagemVitoria.textContent = mensagem;

}

\_esconderVencedor() {

this.\_mensagemVitoria.textContent = '';

}

}

export default JogoDaVelhaUI;

estilos.css

css

Copy code

.tabuleiro {

display: flex;

flex-wrap: wrap;

width: 300px;

height: 300px;

margin: 0 auto;

border: 1px solid black;

}

.casa {

width: 33.333%;

height: 100px;

font-size: 5em;

font-weight: bold;

border: none;

background-color: white;

}

.casa:hover {

cursor: pointer;

background-color: #efefef;

}

.mensagem-vitoria {

margin-top: 10px;

text-align: center;

font-size: 2em;

}

.botao-reiniciar {

margin-top: 10px;

padding: 10px;

font-size: 1.2em;

font-weight: bold;

background-color: white;

border: 1px solid black;

cursor: pointer;

}

Rogério Soares (Merovingio)

continue o código

Para completar o código, vamos agora criar o script principal index.js que vai instanciar as classes Partida e JogoDaVelhaUI e fazer a ligação entre elas:

javascript

Copy code

import Partida from './partida.js';

import JogoDaVelhaUI from './jogoDaVelhaUI.js';

const partida = new Partida();

const jogoDaVelhaUI = new JogoDaVelhaUI(partida);

jogoDaVelhaUI.iniciar();

Por fim, vamos criar um arquivo index.html para importar o script principal e o arquivo de estilos:

html

Copy code

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Jogo da Velha</title>

<link rel="stylesheet" href="estilos.css">

</head>

<body>

<div class="tabuleiro"></div>

<div class="mensagem-vitoria"></div>

<button class="botao-reiniciar"></button>

<script type="module" src="index.js"></script>

</body>

</html>

Com isso, nosso jogo da velha está pronto para ser executado no navegador!

Regenerate response

# Trabalhando com APIs RESTful em JavaScript

# Integração com serviços de terceiros (Google Maps, Facebook, Twitter, etc.)

# Manipulação de arquivos e dados locais em JavaScript

# Trabalhando com WebGL e gráficos 3D em JavaScript

# Desenvolvimento de jogos em JavaScript

# Uso de WebSockets e WebRTC em JavaScript

# Desenvolvimento de aplicações offline em JavaScript

# Otimização de performance em JavaScript

# Uso de bibliotecas e frameworks (jQuery, Angular, React, Vue, etc.)

# Desenvolvimento de extensões para navegadores em JavaScript

# Trabalhando com bancos de dados NoSQL em JavaScript

# Desenvolvimento de aplicações mobile com frameworks híbridos (React Native, Ionic, etc.)

# Segurança em aplicações web com JavaScript

# Desenvolvimento de chatbots em JavaScript

# Uso de tecnologias como WebAssembly em JavaScript

# Desenvolvimento de aplicações de realidade virtual e aumentada com JavaScript

# Trabalhando com Blockchain em JavaScript

# Uso de Machine Learning em JavaScript

# Desenvolvimento de aplicações de reconhecimento de voz em JavaScript

# Desenvolvimento de assistentes virtuais com JavaScript

# Desenvolvimento de aplicações para IoT com JavaScript

**Lista inicial:**

Programação orientada a objetos em JavaScript

Manipulação de eventos e callbacks

Trabalhando com Promises e Async/Await

Programação Funcional em JavaScript

Closures e escopo em JavaScript

Tipos de dados avançados em JavaScript (Map, Set, WeakMap, WeakSet)

Desenvolvimento de aplicações SPA (Single Page Application)

Testes unitários e testes de integração em JavaScript

Uso de módulos em JavaScript (CommonJS, AMD, ES6)

Trabalhando com APIs RESTful em JavaScript

Integração com serviços de terceiros (Google Maps, Facebook, Twitter, etc.)

Manipulação de arquivos e dados locais em JavaScript

Trabalhando com WebGL e gráficos 3D em JavaScript

Desenvolvimento de jogos em JavaScript

Uso de WebSockets e WebRTC em JavaScript

Desenvolvimento de aplicações offline em JavaScript

Otimização de performance em JavaScript

Uso de bibliotecas e frameworks (jQuery, Angular, React, Vue, etc.)

Desenvolvimento de extensões para navegadores em JavaScript

Trabalhando com bancos de dados NoSQL em JavaScript

Desenvolvimento de aplicações mobile com frameworks híbridos (React Native, Ionic, etc.)

Segurança em aplicações web com JavaScript

Desenvolvimento de chatbots em JavaScript

Uso de tecnologias como WebAssembly em JavaScript

Desenvolvimento de aplicações de realidade virtual e aumentada com JavaScript

Trabalhando com Blockchain em JavaScript

Uso de Machine Learning em JavaScript

Desenvolvimento de aplicações de reconhecimento de voz em JavaScript

Desenvolvimento de assistentes virtuais com JavaScript

Desenvolvimento de aplicações para IoT com JavaScript