Téc em Desenvolvimento
de Sistemas Bilíngue

Desenvolver Código Orientado a Objetos

UC4 | Prof. Leonardo de Souza

Interface

Uma interface é uma maneira de definir um contrato (um conjunto de regras) para objetos. Ela especifica quais propriedades ou métodos um objeto deve ter para ser considerado do tipo daquela interface.

Em palavras mais simples, uma interface descreve a estrutura do objeto, o que significa que descreve como o objeto deve se parecer.

Interface

São uma ferramenta poderosa para garantir que tenhamos padrões que sejam seguidos no desenvolvimento de software.

Interface

Embora seja possível criar classes sem usar interfaces, incorporar interfaces em seu código pode trazer benefícios significativos em termos de flexibilidade, compatibilidade, abstração e documentação, tornando seu código mais robusto, fácil de entender e manter.

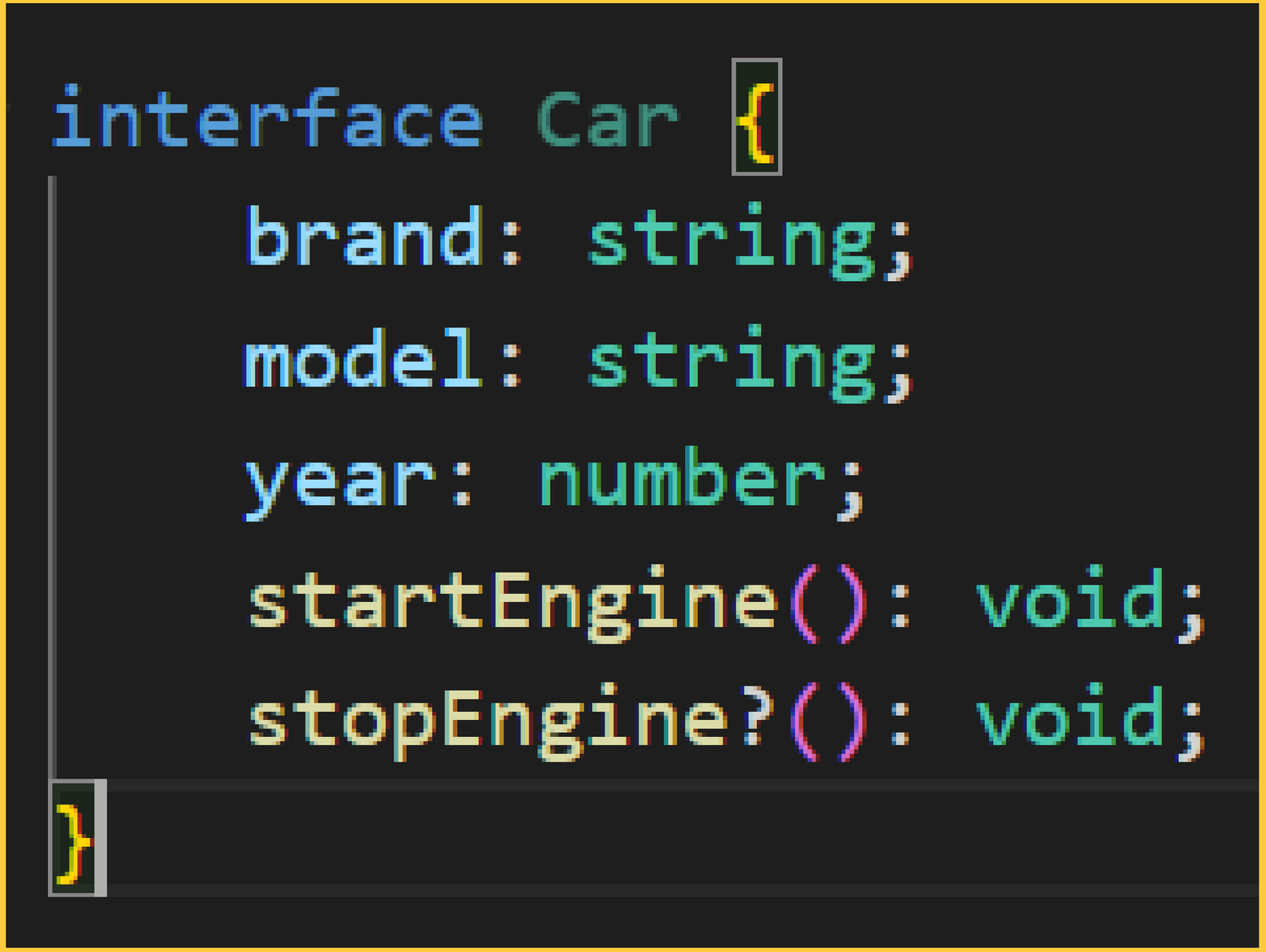
Interfaces

Estrutura básica

Estrutura básica

Uma interface é definida usando a palavra-chave **interface** seguida pelo nome da interface e sua estrutura, que inclui propriedades e métodos.

Interfaces também podem definir métodos, que especificam a assinatura do método sem fornecer sua implementação.


interface Car {
brand: string;
model: string;
year: number;
startEngine(): void;
stopEngine?(): void;
}

Interfaces

Implementação

Implementação

Uma classe pode implementar uma interface usando a palavra-chave **implements** seguida pelo nome da interface.

Ao implementar uma interface, a classe deve fornecer implementações para **todos os métodos exigidos** pela interface e garantir que as **propriedades tenham os tipos corretos.** Assim, ela segue o que chamamos de **contrato.**

```
class Toyota implements Car {
    brand: string;
    model: string;
    year: number;

    constructor(brand: string, model: string, year: number) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    startEngine(): void {
        console.log("Engine started!");
    }
}
```

Interfaces

Interfaces em funções

Interfaces em funções

Interfaces podem ser usadas para definir tipos de parâmetros e retorno de funções.

Usar interfaces em funções torna o código mais flexível e reutilizável, pois permite que as funções aceitem uma variedade de tipos de objetos que seguem o contrato da interface.

```
function displayCarInfo(car: Car): void {
    console.log(`Brand: ${car.brand}, Model: ${car.model}, Year: ${car.year}`);
}
```

Interfaces

Interfaces em objetos literais

Interfaces em objetos literais

Interfaces podem ser usadas para descrever a estrutura de objetos literais, garantindo consistência nos dados fornecidos.

Ao usar interfaces em objetos literais, podemos garantir que os objetos tenham as propriedades corretas e nos campos corretos, ajudando a evitar erros de programação.

```ts
interface Point {
    x: number;
    y: number;
}

function createPoint(x: number, y: number): Point {
    return { x, y };
}

const point = createPoint(10, 20);
console.log(`(${point.x}, ${point.y})`);
```

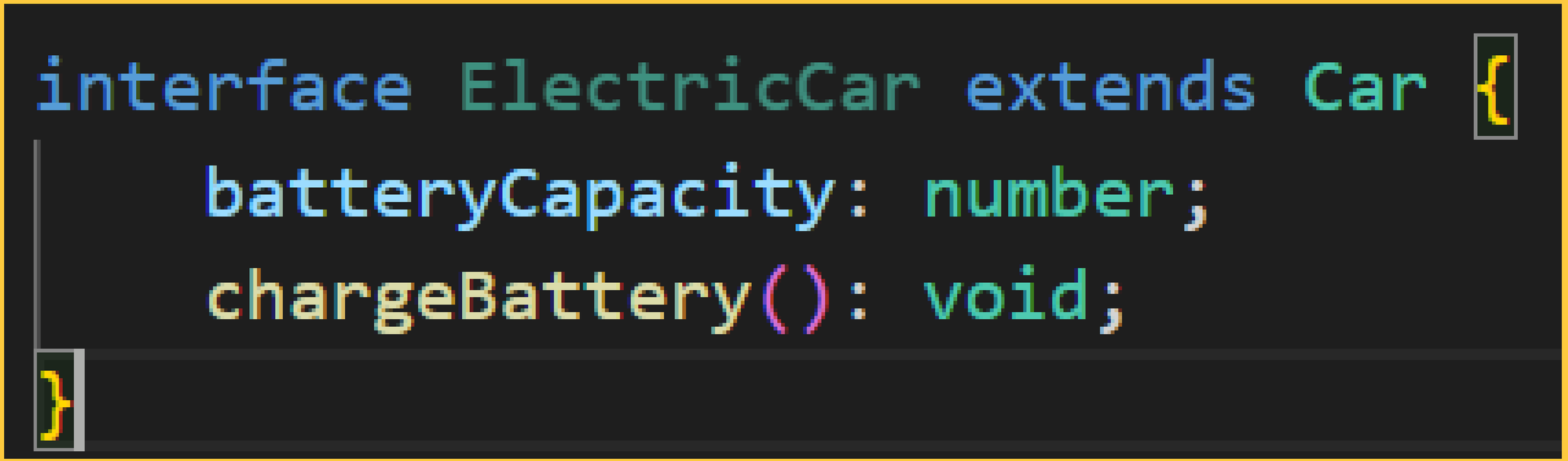
Interfaces

Extensão

Extensão

Uma interface pode estender uma ou mais interfaces usando a palavra-chave **extends**. Isso permite adicionar novas propriedades e métodos à interface estendida.

Ao estender uma interface, podemos adicionar novas propriedades e métodos à interface filha, mantendo a compatibilidade com a interface pai.


interface ElectricCar extends Car {
batteryCapacity: number;
chargeBattery(): void;
}

Interfaces

Interfaces em classes

Interfaces em classes

Interfaces podem ser implementadas por classes para garantir que a classe forneça as funcionalidades especificadas pela interface.

Ao implementar uma interface em uma classe, a classe deve fornecer implementações para todos os métodos exigidos pela interface.

```typescript
interface Shape {
    calculateArea(): number;
}


class Circle implements Shape {
    private radius: number;

    constructor(radius: number) {
        this.radius = radius;
    }

    calculateArea(): number {
        return Math.PI * this.radius ** 2;
    }
}
```

Interfaces

Por que usar?

Por que usar?

Elas ajudam a evitar erros de digitação e a fornecer uma documentação clara sobre como usar um objeto.

Por que usar?

Digamos que estamos construindo um aplicativo de entregas. Precisamos de diferentes veículos para entregar os pacotes: carros, motos e bicicletas. Cada veículo pode ser construído de maneira diferente por dentro, mas todos devem ter um método para se mover.

Por que usar?

Usando interfaces, podemos criar um acordo comum chamado "Veículo". Assim, mesmo que as classes para carros, motos e bicicletas sejam diferentes por dentro, enquanto cada uma delas tiver um método chamado "mover", podemos tratar todos eles da mesma forma quando precisarmos que se movam.

Por que usar?

Isso nos dá flexibilidade porque podemos trocar um veículo por outro, contanto que ambos sigam o mesmo acordo (interface). Então, se precisarmos mudar de uma bicicleta para um carro, por exemplo, podemos fazer isso sem causar problemas em outras partes do nosso código.

Por que usar?

O "acordo" , que chamamos de **contrato**, é essencialmente uma especificação sobre como uma classe deve se comportar. Quando definimos uma interface em TypeScript, estamos criando esse acordo, estabelecendo quais métodos e propriedades uma classe que implementa essa interface deve fornecer e como eles devem funcionar.

Por que usar?

Por exemplo, ao definir uma interface chamada **Vehicle** com um método **move()**, estamos dizendo que qualquer classe que implemente essa interface deve ter um método chamado move() que permita ao veículo se mover.

Isso cria uma expectativa comum em todo o código que interage com objetos que seguem essa interface: eles podem chamar o método move() e esperar que funcione de acordo com o contrato estabelecido pela interface.

Exemplo

Vamos considerar um exemplo mais complexo para ilustrar por que as interfaces são úteis. Vamos criar um sistema de loja online com diferentes tipos de produtos e métodos de pagamento.

Exemplo

Primeiro, vamos definir uma interface **Product** para representar o contrato básico que todos os produtos devem seguir:

```ts
// product.ts
export interface Product {
    name: string;
    price: number;

    /*Essa declaração especifica que qualquer classe
    que implemente a interface Product deve ter um método
    chamado getDescription que retorna uma string.*/
    getDescription(): string;
    /* Se você não criar esse método dentro da classe que
    implementar esta interface, o TypeScript irá gerar um erro de compilação.
}
```

Exemplo

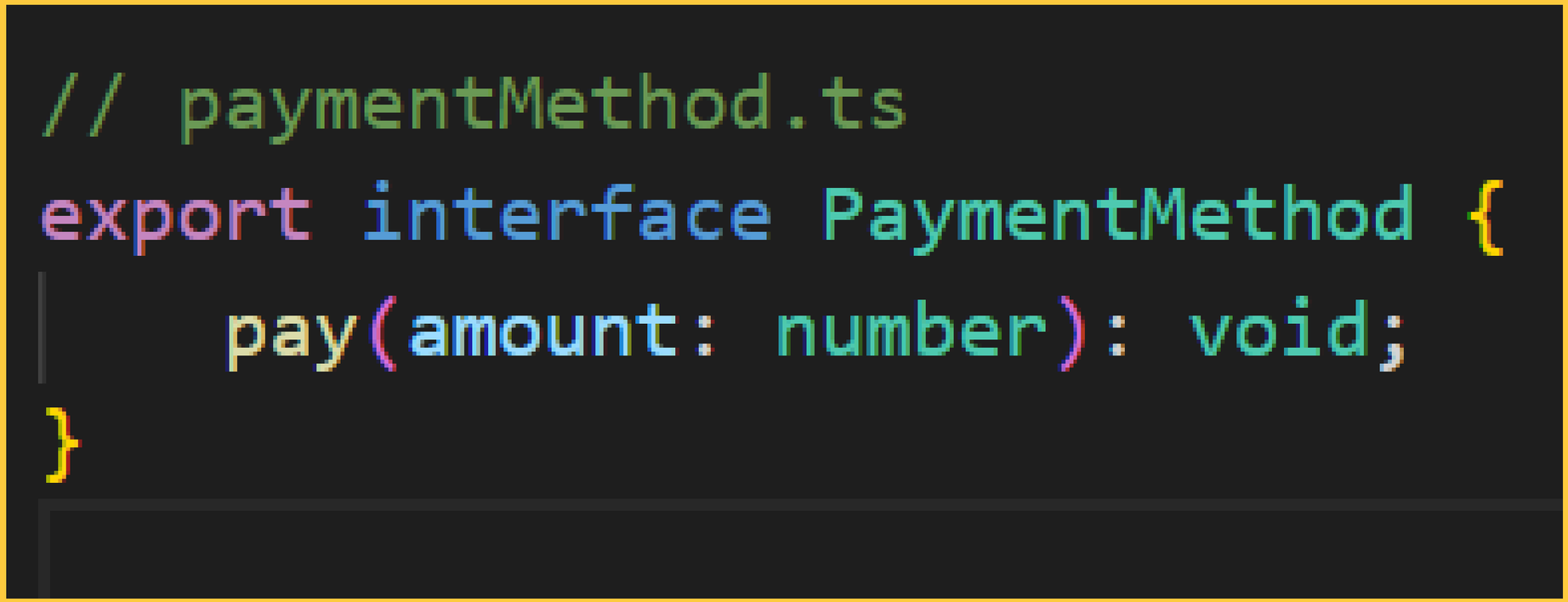
Agora, vamos implementar duas classes de produtos diferentes que seguem essa interface: **Book** e **Electronics**.

```ts
// book.ts
import { Product } from './product';

export class Book implements Product {
    name: string;
    price: number;
    author: string;

    constructor(name: string, price: number, author: string) {
        this.name = name;
        this.price = price;
        this.author = author;
    }

    getDescription(): string {
        /*é um método que formata o valor numérico this.price para exibir duas casas decimais
        após a vírgula, transformando-o em uma string.*/
        return `Livro: ${this.name} - Autor: ${this.author} - Preço: R$${this.price.toFixed(2)}`;
    }
}
```

```ts
// electronics.ts
import { Product } from './product';

// Definição da classe Electronics implementando a interface Product
export class Electronics implements Product {
    // Propriedades da classe
    name: string;
    price: number;
    brand: string;

    // Construtor da classe
    constructor(name: string, price: number, brand: string) {
        this.name = name;
        this.price = price;
        this.brand = brand;
    }

    // Implementação do método getDescription fornecido pela interface Product
    getDescription(): string {
        return `Eletrônico: ${this.name} - Marca: ${this.brand} - Preço: R$${this.price.toFixed(2)}`;
    }
}
```

Exemplo

Agora, vamos definir uma interface **PaymentMethod** para representar o contrato básico que todos os métodos de pagamento devem seguir:


// paymentMethod.ts
export interface PaymentMethod {
pay(amount: number): void;
}

Exemplo

Vamos implementar duas classes de métodos de pagamento diferentes que seguem essa interface: **CreditCardPayment** e **DigitalWalletPayment**.

```ts
// creditCardPayment.ts
import { PaymentMethod } from './paymentMethod';

export class CreditCardPayment implements PaymentMethod {
    pay(amount: number): void {
        console.log(`Pagamento de R$${amount.toFixed(2)} realizado com cartão de crédito.`);
    }
}
```

```ts
// digitalWalletPayment.ts
import { PaymentMethod } from './paymentMethod';

export class DigitalWalletPayment implements PaymentMethod {
    pay(amount: number): void {
        console.log(`Pagamento de R$${amount.toFixed(2)} realizado com carteira digital.`);
    }
}
```

Exemplo

Agora, vamos criar uma classe **ShoppingCart** que pode conter uma variedade de produtos e permite que os clientes paguem com diferentes métodos de pagamento.

```ts
// shoppingCart.ts
import { Product } from './product';
import { PaymentMethod } from './paymentMethod';

// Definição da classe ShoppingCart
export class ShoppingCart {
    // Propriedades da classe
    private items: Product[] = []; //array de objetos do tipo Project
    private paymentMethod: PaymentMethod; //propriedade do tipo PaymentMethod

    // Construtor da classe
    constructor(paymentMethod: PaymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    // Método para adicionar itens ao carrinho
    addItem(item: Product): void {
        this.items.push(item);
    }
```

```
    // Método para calcular o total
    getTotal(): number {
        /*o que essa linha faz é iterar sobre todos os itens no array
        items e somar o preço de cada item ao total, começando com um total inicial de 0. */
        return this.items.reduce((total, item) => total + item.price, 0);
    }

    // Método para realizar o checkout
    checkout(): void {
        const total = this.getTotal();
        this.paymentMethod.pay(total);
    }
}
```

Exemplo

Por fim, vamos usar todas essas classes juntas em nosso arquivo principal (que pode ser o **index.ts**)

```ts
// index.ts
import { Book } from './book';
import { Electronics } from './electronics';
import { CreditCardPayment } from './creditCardPayment';
import { DigitalWalletPayment } from './digitalWalletPayment';
import { ShoppingCart } from './shoppingCart';

// Criando alguns produtos
const livro = new Book("A Arte da Guerra", 29.99, "Sun Tzu");
const celular = new Electronics("Smartphone", 999.99, "Apple");

// Criando métodos de pagamento
const cartaoCredito = new CreditCardPayment();
const carteiraDigital = new DigitalWalletPayment();

// Criando um carrinho de compras e adicionando produtos
const carrinho = new ShoppingCart(cartaoCredito);
carrinho.addItem(livro);
carrinho.addItem(celular);

// Finalizando a compra
carrinho.checkout();
```

Neste exemplo, podemos ver claramente como as interfaces facilitam a interoperabilidade entre diferentes partes do sistema.

1

As interfaces **Product** e **PaymentMethod** estabelecem contratos claros sobre como os produtos devem ser e como os métodos de pagamento devem funcionar.

2

As classes **Book**, **Electronics**, **CreditCardPayment** e **DigitalWalletPayment** implementam essas interfaces, garantindo que elas sigam os contratos estabelecidos.

3

A classe **ShoppingCart** pode lidar com uma variedade de produtos e métodos de pagamento, graças às interfaces, permitindo que o código seja flexível e expansível.