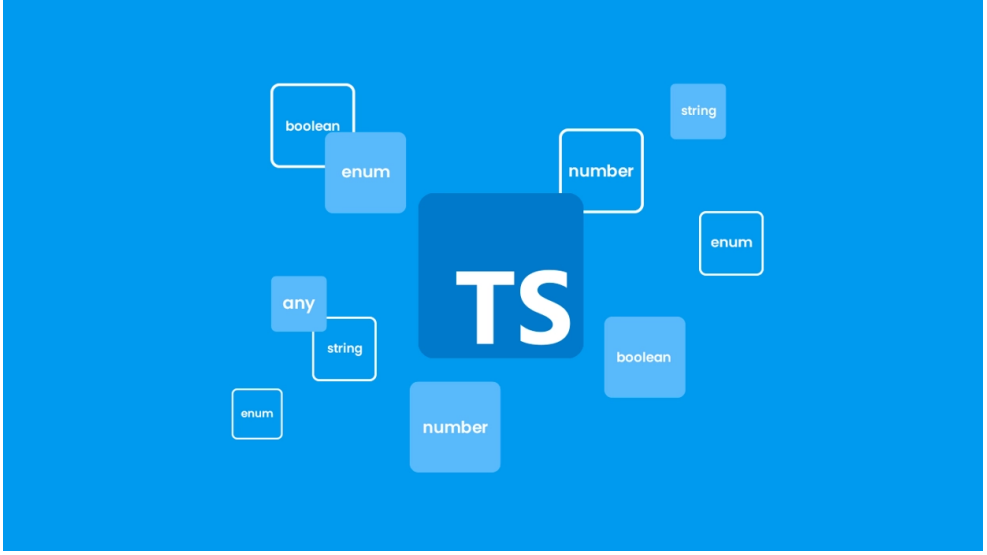
Programação Orientada a Objetos em TypeScript

Este artigo aborda os principais conceitos e práticas da programação orientada a objetos em TypeScript, incluindo classes, herança e interfaces.

Introdução 🚀



A Programação Orientada a Objetos (POO) é uma das metodologias de desenvolvimento de software mais populares no mundo. Ela se concentra na criação de objetos que possuem atributos e métodos, permitindo que os desenvolvedores criem códigos reutilizáveis e modulares. TypeScript, por outro lado, é uma linguagem de programação de código aberto desenvolvida pela Microsoft que amplia a sintaxe do JavaScript, adicionando recursos como tipos estáticos e interfaces. Este artigo abordará a Programação Orientada a Objetos em TypeScript, seus conceitos e práticas.

Conceitos Básicos 📚

Em POO, um objeto é uma instância de uma classe que pode conter variáveis de instância, métodos e construtores. Uma classe, por sua vez, é uma entidade que define as propriedades e comportamentos de um objeto. Em TypeScript, a definição de uma classe é feita da seguinte forma:

```
class Carro {
  marca: string;
  modelo: string;
  ano: number;
```

```
    constructor(marca: string, modelo: str
      this.marca = marca;
      this.modelo = modelo;
      this.ano = ano;
    }

    acelerar() {
      console.log(`NULL NULL está aceleran
    }

    frear() {
      console.log(`NULL NULL está freando!
    }
  }
```

Neste exemplo, temos uma classe `Carro` que possui três variáveis de instância (`marca`, `modelo` e `ano`), um construtor e dois métodos (`acelerar` e `frear`). O construtor é responsável por inicializar as variáveis de instância e os métodos definem o comportamento do objeto.

Encapsulamento 📦

Encapsulamento é um dos princípios fundamentais da POO e refere-se à capacidade de esconder informações dentro de uma classe. Em TypeScript, isso é feito através do uso de modificadores de acesso, que podem ser `public`, `private` ou `protected`. Um membro `public` é acessível a partir de qualquer lugar, enquanto um membro `private` só pode ser acessado dentro da própria classe. Um membro `protected` é semelhante a um membro `private`, mas pode ser acessado dentro de classes derivadas.

```
class ContaBancaria {
  private saldo: number;

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

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

  sacar(valor: number) {
    if (valor > this.saldo) {
      console.log("Saldo insuficiente!")
    } else {
      this.saldo -= valor;
    }
  }
}
```

Neste exemplo, temos uma classe `ContaBancaria` que possui uma variável de instância `saldo` com modificador `private`. Isso significa que o saldo só pode ser acessado dentro da própria classe. Os métodos `depositar` e `sacar` são responsáveis por manipular o saldo.

Herança 👨‍👨‍👦

Herança é outro princípio importante da POO e refere-se à capacidade de criar novas classes a partir de classes existentes. A classe que é estendida é chamada de classe pai ou superclasse, enquanto a classe que estende é chamada de classe filha ou subclasse. Em TypeScript, isso é feito através da

palavra-chave `extends`.

```
class Animal {
  nome: string;

  constructor(nome: string) {
    this.nome = nome;
  }

  mover(distancia: number = 0) {
    console.log(`NULL moveu NULL metros`
  }
}

class Cachorro extends Animal {
  latir() {
    console.log("Au au!");
  }
}

const cachorro = new Cachorro("Rex");
cachorro.mover(10); // Rex moveu 10 metr
cachorro.latir(); // Au au
!
```

Neste exemplo, temos uma classe `Animal` que possui uma variável de instância `nome` e um método `mover`. A classe `Cachorro` estende a classe `Animal` e adiciona um método `latir`. O objeto `cachorro` é uma instância da classe `Cachorro` e, portanto, herda os métodos e variáveis da classe `Animal`.

Polimorfismo 🤝

Polimorfismo é a capacidade de uma classe filha substituir um método da classe pai. Isso permite que diferentes objetos sejam tratados de maneira semelhante, mesmo que tenham comportamentos diferentes. Em TypeScript, isso é feito através do uso da palavra-chave `override`.

```
class Animal {
  nome: string;

  constructor(nome: string) {
    this.nome = nome;
  }

  mover(distancia: number = 0) {
    console.log(`NULL moveu NULL metros`
  }
}

class Cachorro extends Animal {
  mover(distancia: number = 0) {
    console.log(`NULL correu NULL metros
  }

  latir() {
    console.log("Au au!");
  }
}

const animal: Animal = new Animal("Anima
const cachorro: Animal = new Cachorro("R

animal.mover(10); // Animal moveu 10 met
cachorro.mover(10); // Rex correu 10 met
```

Neste exemplo, temos uma classe `Animal` com um método `mover`. A classe `Cachorro` estende a classe `Animal` e sobrescreve o método `mover`. O objeto `animal` é uma instância da classe `Animal` e o objeto `cachorro` é uma instância da classe `Cachorro`. Ao chamar o método `mover`, cada objeto imprime uma mensagem diferente, mostrando que mesmo que sejam tratados como objetos da classe pai, eles têm comportamentos diferentes.

Interfaces 🌐

Interfaces são contratos que definem os membros que uma classe deve implementar. Em TypeScript, isso é feito através da palavra-chave `interface`.

```
interface Veiculo {
  marca: string;
  modelo: string;
  ano: number;
  acelerar(): void;
  frear(): void;
}

class Carro implements Veiculo {
  marca: string;
  modelo: string;
  ano: number;

  constructor(marca: string, modelo: str
    this.marca = marca;
    this.modelo = modelo;
    this.ano = ano;
  }

  acelerar() {
```

```
    console.log(`NULL NULL está aceleran
  }

  frear() {
    console.log(`${this .marca} NULL est
  }
}

const carro = new Carro("Fiat", "Uno", 2
carro.acelerar(); // Fiat Uno está acele
!carro.frear(); // Fiat Uno está freando
!
```

Neste exemplo, temos uma interface `Veiculo` que define as propriedades e métodos que uma classe deve ter para ser considerada um veículo. A classe `Carro` implementa a interface `Veiculo`, garantindo que tenha todas as propriedades e métodos definidos na interface.

Genéricos 💊

Genéricos são tipos que permitem a criação de funções e classes que funcionam com diferentes tipos de dados, sem a necessidade de definir o tipo específico antecipadamente. Em TypeScript, isso é feito através do uso de tipos genéricos.

```
function imprimir<T>(valor: T) {
  console.log(valor);
}

imprimir<string>("Olá, mundo!"); // Olá,
!imprimir<number>(42); // 42
imprimir<boolean>(true); // true
```

Neste exemplo, temos uma função `imprimir` que usa um tipo genérico `T`. Isso permite que a função seja usada com diferentes tipos de dados, sem a necessidade de definir o tipo antecipadamente.

Conclusão 📚

A programação orientada a objetos é uma abordagem popular na construção de software e TypeScript oferece suporte completo para esse paradigma. Neste artigo, discutimos os principais conceitos da POO em TypeScript, incluindo classes, herança, polimorfismo, interfaces e genéricos. É importante lembrar que, embora a POO possa ser uma abordagem poderosa, ela não é a única maneira de escrever software e deve ser usada com cuidado e consideração em cada projeto.

Comentários

Nome:

Email (não será publicado):

Comentário:

Enviar Comentário