BACHARELADO EM SISTEMAS DE INFORMAÇÃO
CEFET-RJ NOVA FRIBURGO

Programação de Clientes Web

PROF. THIAGO DELGADO PINTO
thiago_dp (at) yahoo (dot) com (dot) br

POO em TypeScript

versão: 2020.09.23

AGENDA

- classes e objetos
- atributos
- encapsulamento
- inicialização de atributos
- métodos
- propriedades
- construtor e destrutor
- sobrecarga
- herança e polimorfismo
- sobrescrita
- métodos estáticos
- atributos estáticos
- métodos abstratos
- métodos finais
- classes finais
- interfaces
- classes anônimas

POO

introdução

vamos nos concentrar na **sintaxe** de TypeScript 4 para **POO**

não veremos a sintaxe completa

- consulte o guia oficial para isso

não discutiremos a fundo os **conceitos** de POO

- já foram tratados em disciplinas anteriores
- porém, tire suas dúvidas

classes e objetos

```
class Gato {
}

const g = new Gato();
console.log( g ); // Gato {}
```

atributos

```
class Gato {
  public nome: string = ''; // Requer valor inicial
  public idade?: number; // Default undefined
}

const g = new Gato();
console.log( g ); // Gato { nome: '' }
g.nome = 'Tom';
g.idade = 3;
console.log( g ); // Gato { nome: 'Tom', idade: 3 }
```

encapsulamento

Acesso	`private`	`protected`	`public`
Dentro da classe	✓	✓	✓
Em classe filha	-	✓	✓
Fora da classe	-	-	✓

métodos


```
class Gato {
  som(): string { // visibilidade pública por padrão
    return 'miau!';
  }
}

const g = new Gato();
console.log( g.som() ); // miau!
```

métodos


```
class Gato {
  public som(): string { // com visibilidade
    return 'miau!';
  }
}

const g = new Gato();
console.log( g.som() ); // miau!
```

acesso à instância atual da classe


```
this.
```

acesso à instância atual da classe


```
class Inimigo {
  private pontosVida: number = 100;
  public sofrerDanoPequeno(): void {
    this.pontosVida -= 5;
  }
  public sofrerDanoGrande(): void {
    this.pontosVida -= 20;
  }
  public energia(): number {
    return this.pontosVida;
  }
  public vivo(): boolean {
    return this.energia() > 0;
  }
}
```

```
const i = new Inimigo();
console.log( i.energia() ); // 100
i.sofrerDanoPequeno(); // 95
i.sofrerDanoGrande(); // 75
console.log( i.vivo()
  ? 'Vivo': 'Morto' ); // Vivo
```

métodos com argumentos valorados

```
class Jogador {
  private energia: number = 100;
  private mana: number = 50;
  public getEnergia(): number { return this.energia; }
  public getMana() : number { return this.mana; }
  public sofrerDano( valor: number = 10 ): void {
    this.energia -= valor;
  }
  public usarMana( valor: number = 5 ): void {
    this.mana -= valor;
  }
}

const j = new Jogador();
console.log( 'HP', j.getEnergia(), ', MP', j.getMana() ); // HP 100, MP 50
j.sofrerDano(); // tira 10
j.sofrerDano( 15 ); // tira 15
j.usarMana(); // usa 5
```

propriedades

```
class Jogador {
  private _energia: number = 100;
  private _mana: number = 50;
  public get energia(): number { return this._energia; }
  public set energia( valor: number ) {
    if ( valor < 0 || valor > 100 ) { return; }
    this._energia = valor;
  }
  public get mana(): number { return this._mana; }
  public set mana( valor: number ) {
    if ( valor < 0 || valor > 100 ) { return; }
    this._mana = valor;
  }
}

const j = new Jogador();
j.energia -= 25; // dano
j.mana += 10; // recuperação
console.log( j ); // Jogador { _energia: 75, _mana: 60 }
```

funções set **não** tem tipo de retorno

construtor e destrutor

em TypeScript não há destrutor, só construtor

```
class Radio {
  public volume: number = 50;
  public constructor() {
    console.log( 'ligado com volume', this.volume );
  }
}

new Radio(); // ligado com volume 50
```

construtor com argumentos

```
class Radio {
  private ligado?: boolean;
  private volume?: number;
  public constructor( ligado: boolean, volume: number ) {
    this.ligado = ligado;
    this.volume = volume;
  }
  public getLigado(): boolean {
    return this.ligado;
  }
  public getVolume(): number {
    return this.volume;
  }
}

// const r0 = new Radio();          // Erro – requer argumentos
// const r1 = new Radio( true );    // Erro – requer segundo argumento
const r2 = new Radio( true, 10 ); // OK
console.log( r2.getVolume() );    // 10
```

construtor com argumentos valorados

```
class Radio {
  private ligado?: boolean;
  private volume?: number;
  public constructor( ligado: boolean = true, volume: number = 50 ) {
    this.ligado = ligado;
    this.volume = volume;
  }
  public getLigado(): number { return this.ligado; }
  public getVolume(): number { return this.volume; }
}


const r0 = new Radio();             // OK, ligado    e com volume 50
const r1 = new Radio( false );      // OK, desligado e com volume 50
const r2 = new Radio( true, 10 ); // OK, ligado    e com volume 10
console.log( r2.getVolume() ); // 10
```

construtor com atributos

TypeScript permite declarar atributos **diretamente** no construtor

```
class Radio {
  public constructor(
    private ligado: boolean,
    private volume: number
  ) {
  }
  public getLigado(): number { return this.ligado; }
  public getVolume(): number { return this.volume; }
}

// const r0 = new Radio(); // Erro – argumentos requeridos
const r1 = new Radio( true, 10 );  // ligado e com volume 10
console.log( r1.getVolume() );     // 10
```

construtor com atributos valorados


```
class Radio {
  public constructor(
    private ligado: boolean = true,
    private volume: number = 50
  ) {
  }
  public getLigado(): number { return this.ligado; }
  public getVolume(): number { return this.volume; }
}

const r = new Radio();          // ligado e com volume 50
console.log( r.getVolume() ); // 50
```

construtor com atributos valorados


```
class Radio {
  private frequencia: number = 88.1; // Só funciona nessa frequência
  public constructor(
    private ligado: boolean = true,
    private volume: number = 50
  ) {
  }
  public getLigado(): number { return this.ligado; }
  public getVolume(): number { return this.volume; }
  public getFrequencia(): number { return this.frequencia; }
}

const r = new Radio();              // ligado e com volume 50
console.log( r.getFrequencia() ); // 88.1
```

construtor com atributos somente leitura

```
class Radio {
  public constructor(
    public readonly ligado: boolean = true,
    public readonly volume: number = 50,
    public readonly frequencia: number = 88.1,
  ) {
  }
}

const r = new Radio(); // ligado, vol 50, 88.1
console.log( r.frequencia ); // 88.1
r.volume = 20; // Erro – somente leitura
```

sobrecarga de métodos


```
class Radio {
  /* ... */
  public setFrequencia( f: number ): void; // Assinatura 1
  public setFrequencia( f: string ): void; // Assinatura 2
  // Implementação com assinatura compatível com ambas
  public setFrequencia( f: string | number ): void {
    if ( 'number' === typeof f ) {
        this.frequencia = f;
    } else if ( 'string' === typeof f
      && ! Number.isNaN( f ) && Number.isFinite( f ) ) {
        this.frequencia = parseFloat( f );
    }
  }
}
```

Tipo de *f* poderia ser *any*

sobrecarga de métodos – alternativa usual


```
class Radio {
  /* ... */
  // Assinatura única, com alternativas de parâmetros e tipos
  public setFrequencia( f: string | number ): void {
    if ( 'number' === typeof f ) {
        this.frequencia = f;
    } else if ( 'string' === typeof f
      && ! Number.isNaN( f ) && Number.isFinite( f ) ) {
        this.frequencia = parseFloat( f );
    }
  }
}
```

herança e polimorfismo


```
class MagoInimigo extends Inimigo {
  public magiaSimples(): string { return 'Estupefaça'; }
  public danoMagiaSimples(): number { return 10; }
  public magiaComplexa(): string { return 'Confrigo'; }
  public danoMagiaComplexa(): number { return 25; }
}

function imprimirEnergia( i: Inimigo ): void {
  console.log('Está ', i.vivo() ? 'vivo':'morto', ', com ', i.energia(), ' de energia.');
}

const m = new MagoInimigo();
m.magiaSimples();     // Estupefaça
m.sofrerDanoGrande(); // -20
imprimirEnergia( m );  // Está vivo, com 80 de energia

const i = new Inimigo();
imprimirEnergia( i );  // Está vivo, com 100 de energia
```

TypeScript permite herdar de
apenas uma classe

sobrescrita

```
class MagoSeniorInimigo extends MagoInimigo {
  public magiaSimples(): string { return 'Confrigo'; }
  public danoMagiaSimples(): number { return 25; }
  public magiaComplexa(): string { return 'Avada Kedavra'; }
  public danoMagiaComplexa(): number { return 50; }
}

function imprimirMagias( m: MagoInimigo ) {
  console.log( m.magiaSimples(), ' com dano ', m.danoMagiaSimples() );
  console.log( m.magiaComplexa(), ' com dano ', m.danoMagiaComplexa() );
}

const m = new MagoInimigo();
imprimirMagias( m ); // Estupefaça com dano 10\nConfrigo com dano 25
const ms = new MagoSeniorInimigo();
imprimirMagias( ms ); // Confrigo com dano 25\nAvada Kedabra com dano 50
```

acesso à implementação da classe pai

`super`

acesso à implementação da classe pai


```
class Jogador {
  public constructor(
    private energia: number = 100,
    private mana: number = 50
  ) {}
  /* ... */
}

class Guerreiro extends Jogador {
  public constructor(
    energia: number = 100,
    mana: number = 50
    private subclasse: string = 'Aprendiz'
  ) {
    super( energia, mana ); // deve ser a primeira chamada do construtor
  }
  /* ... */
}
```

acesso à implementação da classe pai


```
class MagoHabilidosoInimigo extends MagoSeniorInimigo {
  /* ... */
  public magiaSimples(): string { // Faz combo
    return super.magiaSimples() + ' + Estupefaça';
  }
  public danoMagiaSimples(): number {
    return super.danoMagiaSimples() + 10;
  }
}

const m = new MagoHabilidosoInimigo();
imprimirMagias( m );
// Confrigo + Estupefaça com dano 35\nAvada Kedabra com dano 50
```

métodos estáticos


```
class Gato {
  public static som(): string {
    return 'miau!';
  }
}

echo Gato.som(); // miau!
```

acesso à algo estático dentro da classe

```
NomeDaClasse.
```

acesso à algo estático dentro da classe


```
class Samurai {
  public constructor( private nome: string ) {}
  public getNome(): string { return this.nome; }
  public static designacao(): string { return 'o samurai'; }
  public apresentacao(): string {
    return this.getNome() + ',' + Samurai.designacao();
  }
}

console.log( Samurai.designacao() ); // o samurai

const s = new Samurai( 'Jack' );
console.log( s.apresentacao() ); // Jack, o samurai
```

atributos estáticos

```
class Impressora {
  private static contagemImpressoes: number = 0;
  public static impressoes(): number {
    return Impressora.contagemImpressoes;
  }
  public imprimir( texto: string = '\n' ) {
    console.log( texto );
    Impressora.contagemImpressoes++;
  }
}

Impressora.impressoes(); // 0
const i = new Impressora();
i.imprimir( 'Teste' );
Impressora.impressoes(); // 1
const i2 = new Impressora();
i2.imprimir( 'Nova linha' );
Impressora.impressoes(); // 2
```

classes e métodos abstratos

```
const PI: number = 3.14;

abstract class Forma {
  public constructor( private x: number = 0, y: number = 0 ) {}
  public getPosicaoX() { return this.x; }
  public getPosicaoY() { return this.y; }
  public abstract area(): number; // sem corpo, deve ser implementado em alguma classe filha
}

class Circunferencia extends Forma {
  public constructor( x: number = 0, y: number = 0, private raio: number = 1 ) {
    super( x, y );
  }
  public getRaio() { return this.raio; }
  public comprimento() { return 2 * PI * this.raio; }
  public area() { return PI * ( this.raio ** 2 ); } // ** é potência em TypeScript 1.7+
}

$c = new Circunferencia( 10 );
echo $c->area(), ' cm'; // 314 cm
```

métodos e classes finais

TypeScript ainda não os suporta – veja Issue relacionada

alternativa:

```
function final(target: Object, key: string|symbol, descriptor: PropertyDescriptor) {
  descriptor.writable = false;
}

class Pai {
  @final
  falar(): void { console.log('Pai'); }
}

class Filho extends Pai {
  // Error: "falar" is read-only
  falar(): void { console.log('Filho'); }
}

const f = new Filho();
f.falar();
```

interfaces


```
interface Animal {
  tipoDeSom(): string;
}

class Gato implements Animal {
  public tipoDeSom(): string { return 'miado'; }
}

class Cao implements Animal {
  public tipoDeSom(): string { return 'latido'; }
}

function mostrarSom( readonly a: Animal ) {
  console.log( a.tipoDeSom() );
}

mostrarSom( new Gato() ); // miado
mostrarSom( new Cao() ); // latido
```

na prática, use uma interface para representar um conjunto de comportamentos que todas as suas instâncias devem ter.

ela serve como um **modelo/contrato sintático** apenas.
é uma convenção, já que não há garantia real de comportamento.

```
interface Impressora {
  imprimir( texto: any ): void;
}

class ImpressoraEmConsole implements Impressora {
  public imprimir( texto: any ): void { console.log( texto ); }
}

class ImpressoraEmArquivo implements Impressora {
  public constructor( private arquivo: string ) {}
  public imprimir( texto: any ): void { /* ... */ }
}

class ImpressoraFalsa implements Impressora {
  public imprimir( texto: any ) {}
}

const impressoras = [ new ImpressoraEmConsole(), new ImpressoraEmArquivo( 'saída.txt' ), new
ImpressoraFalsa() ];
for ( const imp of impressoras ) { imp.imprimir( 'Testando impressão\n' ); }
```

interfaces


```
interface Impressora {  imprimir( texto: any ): void;  }

interface ImpressoraConfiguravel extends Impressora {
  configurar( parametros: Map< string, any > ): void;
}

class ImpressoraFabricanteXPTO {
  public enviar( conteudo: any ) { /*...*/ }
  public definirPorta( porta: string ) { /*...*/ }
}

class ImpressoraXPTO extends ImpressoraFabricanteXPTO implements ImpressoraConfiguravel {
  public imprimir( texto: any ): void { this.enviar( texto ); }
  public configurar( parametros: Map< string, any > ): void {
    this.definirPorta( parametros.get('porta') ); }
}

const i = new ImpressoraXPTO();
i.configurar( new Map( Object.entries( { 'porta': 'USB001' } ) ) );
i.imprimir( 'Teste de Impressão' );
```

é possível implementar ou estender
mais de uma interface

interfaces como moldes de objetos

em TypeScript, interfaces também
podem ser moldes para objetos

interfaces como moldes de objetos


```
interface Contato {
  nome: string;
  telefone?: string; // "?" torna telefone opcional
}

const c1: Contato = { nome: "Alan Key" };
const c2: Contato = {
  nome: "Anders Hejlsberg",
  telefone: "2299999-8888"
};
```

interfaces com atributos imutáveis

```
interface Ponto {
  readonly x: number;
  readonly y: number;
}

let p: Ponto = { x: 10, y: 20 };
p.x = 30; // Erro
// Cannot assign to 'x' because it is a read-only property.
```

observação – `ReadOnlyArray`

TypeScript oferece a classe `ReadonlyArray` para criar ***arrays*** **imutáveis** após a criação

ela é um *template* e deve receber o **tipo** do array

ex.: `let a: ReadonlyArray< string > = [ 'Ana', 'Bia', 'Carla' ];`

ela **não oferece métodos** ou **propriedades** que possam **mudar** o *array* instanciado – ex:

```
a.push( 'Daniela' ); // Erro
a[ 0 ] = 'Amanda'; // Erro
```

interfaces com índices

possibilitam que instâncias da interface possam acessar um **atributo** via índice **inteiro** ou **string**

exemplo:

```
interface Candidatos {
  [indice: number]: string;
}
const c: Candidatos = [ "Ana", "Bia" ];
console.log( c[ 1 ] ); // Bia
```

classes anônimas


```
const calculo = new class {
  public soma( x: number, y: number ): number {
    return x + y;
  }
};
console.log( calculo.soma( 1, 2 ) ); // 3
```

classes anônimas


```
const vaca = new class implements Animal {
  public tipoDeSom(): string {
    return 'mugido';
  }
};
mostrarSom( vaca );

const gatoDoMato = new class extends Gato {
  public tipoDeSom(): string {
    return 'esturro';
  }
};
mostrarSom( gatoDoMato );
```

referências

TypeScript Website. **TypeScript Handbook**. Disponível em: https://www.typescriptlang.org/docs/handbook/intro.html

fim

2020.08.21: Reformulação.