



BACHARELADO EM SISTEMAS DE INFORMAÇÃO
CEFET-RJ NOVA FRIBURGO

Programação de Clientes Web

PROF. THIAGO DELGADO PINTO

thiago.pinto@cefet-rj.br

Tipos em TypeScript

versão: 2020.09.24



Licença Creative Commons 4

AGENDA

tipos básicos do ES

classes derivadas de Object

tipos básicos adicionais do TS

any, unknown, void, never, enum, tupla

moldagem de tipos

tipos alternativos

tipos nomeados

função como tipo

objeto como tipo

tipos básicos do EcmaScript

tipos primitivos	outros	invólucro
string		String
number		Number
bigint		BigInt
boolean		Boolean
null		-
undefined		-
ES6+ ➡ symbol		Symbol
	object	Object

Object é a classe pai de todas as outras classes.

classes derivadas de Object

Array

Function

RegExp

Date

Error

JSON

Math

Map

WeakMap

Set

WeakSet

Proxy

Reflect



ES6+

tipos básicos adicionais do TypeScript

any	Representa qualquer tipo. O compilador deixa de verificar o tipo em atribuições.
unknown	Representa um tipo desconhecido . O compilador só permite a atribuição para outro tipo se for verificado antes (<i>i.e.</i> , com typeof).
void	Representa a ausência de tipo. Geralmente é usado como retorno de uma função.
never	Representa um retorno que nunca ocorre, pois o código nunca alcança o ponto de retorno.
enum	Restringe os valores a um conjunto de identificadores.
tupla	Construção que expressa um <i>array</i> com um número fixo de elementos, cujos tipos são conhecidos, mas não necessariamente iguais.

assumido por **padrão** se tipo não for especificado

útil quando se está migrando de JS

compilador não verifica atribuições

```
function imprimir( conteudo: any ): void {  
    console.log( conteudo );  
}
```

```
let x: any = 5;  
const y: number = x; // OK
```

```
imprimir( x );
```

representa um tipo **desconhecido**

compilador verifica atribuições

só permite a atribuição para outro tipo se for verificado antes (*i.e.*, com **typeof**)


```
function imprimir( conteudo: unknown ): void {  
    console.log( conteudo );  
}
```

```
let x: unknown = 5;
```

```
if ( 'number' === typeof x ) {  
    const y: number = x; // OK  
    imprimir( y );  
}
```

```
const z: number = x; // Erro - tipo unknown não é atribuível ao tipo number  
imprimir( z );
```

representa a **ausência** de tipo

geralmente é usado como **retorno de função**

admite **undefined**

admite **null** quando o *flag* **--strictNullChecks** não estiver ligado

void

```
function imprimir( conteudo: any ): void {  
    console.log( conteudo );  
}
```

```
imprimir( x );
```

representa um retorno que **nunca** ocorre

o código nunca alcança o ponto de retorno

ou seja, ocorre um **loop infinito** ou uma **exceção**

```
function soma( a: number, b: number ): never {  
    throw new Error( 'Soma não executada' );  
    return a + b;  
}
```

```
try {  
    console.log( soma( 10, 20 ) );  
} catch ( e: Error ) {  
    console.log( e.message );  
}
```

```
function oi(): never {  
    while ( true ) {  
        console.log( 'Oi' );  
    }  
}
```

```
oi();
```

restringe valores a um conjunto de identificadores

como constantes em um *namespace*

identificadores podem ter valores **inteiros** ou **string**


por padrão, valores são inteiros começando em zero (**0**)

enum – exemplo 1

```
enum Juros {  
    Simples,  
    Compostos  
}
```

```
console.log( Juros.Simples );    // 0  
console.log( Juros.Compostos ); // 1
```

```
console.log( Juros[ 0 ] ); // Simples  
console.log( Juros[ 1 ] ); // Compostos
```



Acesso como *array* faz obter o identificador como *string*.

enum – exemplo 2

```
enum MesPrimeiroTrimestre {  
    Janeiro = 1,  
    Fevereiro, // 2  
    Marco      // 3  
}
```


enum – exemplo 3

```
enum BasicColors {  
    Red    = 'FF0000',  
    Green  = '00FF00',  
    Blue   = '0000FF'  
}
```

há duas notações: `[]` ou `Array<>`

exemplos:

```
const a: number[] = [ 1, 2, 3 ];
```

```
const b: Array< number > = [ 1, 2, 3 ];
```

```
const c: Array< any > = [ 1, 'Oi', false, null ];
```

```
const d: Date[] = [ new Date(), new Date(2020, 1, 1) ];
```

```
const e: number[][] = [ [ 1, 2 ], [ 3, 4 ] ];
```

```
const f: Array< Array< any > > = [ [1, 'Ana'], [2, 'Bia']];
```

tupla – exemplo 1

```
let x: [ number, string ] = [ 1, 'Joana' ];
```

```
console.log( x[ 0 ] ); // 1
```

```
console.log( x[ 1 ] ); // Joana
```

```
const y: [ number, string ] = [ 'Joana', 1 ]; // Erro!
```

```
let a: Array< [ number, string ] > = [  
  [ 1, 'Ana' ], [ 2, 'Bia' ], [ 3, 'Carla' ] ];
```

```
let b: [ number, boolean ] = [ 1, false ];
```

```
b = [ 2, true ];
```

tupla – exemplo 2

```
function procurar(  
  nome: string,  
  valores: Array< [ number, string ] >  
) : [ number, string ] {  
  for ( const v of valores ) {  
    if ( nome === v[ 1 ] ) {  
      return v;  
    }  
  }  
  throw new Erro( 'Não encontrado.' );  
}
```

moldagem de tipos

há duas sintaxes permitidas: uso de **as** ou **<>**

exemplo do uso de **as**:

```
const valor: unknown = 'Olá';  
const tamanho: number = (valor as string).length;
```

exemplo do uso de **<>**:

```
const valor: unknown = 'Olá';  
const tamanho: number = (<string> valor).length;
```

tipos alternativos

TS oferece o **operador** `|` ("ou unário") para declarações de tipo

exemplo:

```
let a: number | string = 10;  
a = 'Oi'; // OK
```

```
function imprimir( valor: number | string | boolean ): void {  
    if ( 'boolean' === typeof valor ) {  
        console.log( valor ? 'VERDADEIRO' : 'FALSO' );  
        return;  
    }  
    console.log( valor );  
}
```

```
imprimir( true ); // VERDADEIRO  
imprimir( a ); // Oi
```

tipos nomeados

TS permite usar **type** para nomear tipos

exemplo

```
type NumeroOuString = number | string;
let a: NumeroOuString = 10;
a = 'Oi'; // OK
type Codigo = string;
function limparCodigo(c: Codigo): Codigo {
  return c.trim().replace( /\.\-/g, '' );
}
let cod = limparCodigo( 'ABC-123.456' );
```

função como tipo

parâmetros são definidos entre parênteses

tipo de retorno é definido após seta (\Rightarrow)

exemplo:

```
let f: () => void;  
function digaOi(): void {  
  console.log( 'Oi' );  
}  
f = digaOi;  
f(); // Oi
```


função como tipo – exemplo 2

```
let f: (x: number, y: number) => number;
```

```
function somar(x: number, y: number): number {  
    return x + y;  
}
```

```
function subtrair(x: number, y: number): number {  
    return x - y;  
}
```

```
f = somar;  
console.log( f( 20, 5 ) ); // 25  
f = subtrair;  
console.log( f( 20, 5 ) ); // 15
```

função como tipo – exemplo 3

```
type FuncaoCalculo = (x: number, y: number) => number;
```

```
function somar(x: number, y: number): number {  
  return x + y;  
}
```

```
function subtrair(x: number, y: number): number {  
  return x - y;  
}
```

```
let f: FuncaoCalculo = somar;  
console.log( f( 20, 5 ) ); // 25  
f = subtrair;  
console.log( f( 20, 5 ) ); // 15
```

função como tipo – exemplo 4

```
function fabricarCalculo( tipo: string ): (x: number, y: number) => number {  
  switch ( tipo ) {  
    case 'subtrair':  
      return function(x: number, y: number): number {  
        return x - y;  
      };  
    case 'somar':  
      return function(x: number, y: number): number {  
        return x + y;  
      };  
    default: throw new Error( 'Tipo não disponível' );  
  }  
}
```

```
const f = fabricarCalculo( 'somar' );  
f( 20, 5 ); // 25
```

função como tipo – exemplo 5

```
const LIMITE_ENERGIA: number = 100;
```

```
class Jogador {  
  private _aoMudarEnergia: ((anterior: number, nova: number) => void) | undefined;  
  private _energia: number = LIMITE_ENERGIA;  
  get aoMudarEnergia(): ((anterior: number, nova: number) => void) | undefined {  
    return this._aoMudarEnergia;  
  }  
  set aoMudarEnergia( fn: ((anterior: number, nova: number) => void) | undefined ) {  
    this._aoMudarEnergia = fn;  
  }  
  get energia(): number { return this.energia; }  
  set energia(valor: number) {  
    if ( valor > LIMITE_ENERGIA ) { return; }  
    if ( this._aoMudarEnergia ) { this._aoMudarEnergia( this._energia, valor ); }  
    this._energia = valor;  
  }  
}
```

```
const j = new Jogador();  
j.aoMudarEnergia = function( a: number, n: number ): void { console.log( 'De', a, 'para', n ); };  
j.energia = 70; // Imprime "De 100 para 70"  
j.aoMudarEnergia = undefined;  
j.energia = 50; // Não imprime
```

função como tipo – exemplo 6

```
const LIMITE_ENERGIA: number = 100;
```

```
type EventoMudancaEnergia = (anterior: number, nova: number) => void;
```

```
class Jogador {  
  private _aoMudarEnergia: EventoMudancaEnergia | undefined;  
  private _energia: number = LIMITE_ENERGIA;  
  get aoMudarEnergia(): EventoMudancaEnergia | undefined {  
    return this._aoMudarEnergia;  
  }  
  set aoMudarEnergia( fn: EventoMudancaEnergia | undefined ) {  
    this._aoMudarEnergia = fn;  
  }  
  get energia(): number { return this.energia; }  
  set energia(valor: number) {  
    if ( valor > LIMITE_ENERGIA ) { return; }  
    if ( this._aoMudarEnergia ) { this._aoMudarEnergia( this._energia, valor ); }  
    this._energia = valor;  
  }  
}
```

```
const j = new Jogador();  
j.aoMudarEnergia = function( a: number, n: number ): void { console.log( 'De', a, 'para', n ); };  
j.energia = 70; // Imprime "De 100 para 70"  
j.aoMudarEnergia = undefined;  
j.energia = 50; // Não imprime
```

objeto como tipo – exemplo 1

```
let o1: { nome: string };
```

```
o1 = { nome: 'Bia' }; // OK
```

```
o1 = { nome: 'Bia', idade: 29 }; // Erro
```

```
o1 = { nomeCompleto: 'Beatrix Kiddo' }; // Erro
```

```
let o2: { nome: string, idade?: number };
```

```
o2 = { nome: 'Sara' }; // OK
```

```
o2 = { nome: 'Carlos', idade: 65 }; // OK
```

objeto como tipo – exemplo 2

```
type Contato = { nome: string, telefone: string };
```

```
function procurar(  
  nome: string,  
  contatos: Contato[]  
): Contato | null {  
  for ( const c of contatos ) {  
    if ( nome === c.nome ) {  
      return c;  
    }  
  }  
  return null; // Não encontrado  
}
```

observações finais

classes e **interfaces** serão vistas separadamente

referências

TypeScript Website. **TypeScript Handbook**. Disponível em: <https://www.typescriptlang.org/docs/handbook/>

fim

2020.08.24: Versão inicial.



Licença Creative Commons 4

ESTE MATERIAL PERTENCE AO PROFESSOR THIAGO DELGADO PINTO
E ESTÁ DISPONÍVEL SOB A LICENÇA CREATIVE COMMONS VERSÃO 4.
AO SE BASEAR EM QUALQUER CONTEÚDO DELE, POR FAVOR, CITE-O.