



Introdução ao TypeScript

Material das Aulas

Introdução

Redes

O que é?

Quais problemas o TS Resolve?

Quais problemas o TS traz?

Devo sair tipando tudo?

O que é inferência?

Como instalar?

Executando o compilador

Exemplo JS x TS

noEmitOnError

Comparação com GO

Tipos básicos I

index.ts

String

Number

Boolean

Array

Para compilar utilizamos → npm run build

Para executar o arquivo → node build array.js

Tipos básicos II

Tupla

Enum

Any

Unkown

Null e Undefined

Object

Múltiplos tipos com Union types | type, interface

Funções

Void

Never

Type alias

Union Types

Intersection types

- [Juntando níveis](#)
- [Orientação de Objetos I](#)
 - [Classes](#)
 - [Herança](#)
 - [Métodos Static](#)
- [Orientação de Objetos II](#)
 - [Diferenças entre método `public`, `private` e `protected`](#)
 - [Interfaces \(e semelhanças com type alias\)](#)
 - [Usar Type ou Interface](#)
 - [Type Assertion](#)

Material das Aulas

- [Slides](#)

Introdução

Redes

- [Gabriel Aulas](#)
- [Repositório](#)

O que é?

Linguagem

- Podemos executar o JS no navegador ou no servidor
 - Motor V8 através dos navegadores
 - NodeJS que usa esses motores do lado do Cliente.
- O JS não é uma linguagem compilada, é compilada em tempo execução, não tem tipagem e isso pode trazer fragilidades pra nossa aplicação.
- Erros que poderíamos resolver em compilação
- O TS é uma camada extra (superset) para nossa aplicação, vamos ter o código em TS, transformamos/compilamos para JS e esse sim será executado no navegador ou no servidor, pois eles só entendem JS.

Deno

- É bem similar para o NodeJS, a plataforma é o motor que vai executar nosso código em tempo de desenvolvimento, e ele permite a execução de código em

TS de forma direta.

- Ela já roda TS em tempo de execução, sem precisar converter para JS, para fazer a compilação
- Node e Deno são bem parecidos, é uma brincadeira para ordenação do node de forma certa

```
> 'node'.split('')
< ▶ (4) ["n", "o", "d", "e"]
> 'node'.split('').sort()
< ▶ (4) ["d", "e", "n", "o"]
> 'node'.split('').sort().join('')
< "deno"
```

Quais problemas o TS Resolve?

- Resolve a tipagem fraca do JavaScript, e agora o TS garante tipos e contratos para nossas variáveis

Exemplo

- Com JS e, com TS.
- Conseguimos aplicar validações estáticas, sem mandar pra produção erros que passam despercebidos pelo fato da tipagem fraca.
- Garantimos que parâmetros, definições e retornos sigam uma determinada regra.
- Temos erros em tempo de desenvolvimento e não produção

```
1 function soma (a, b) {
2     return a + b;
3 }
4
5 console.log(soma(1, 1)) // 2
6 console.log(soma('1', '1')) // 11
```

```
~/Development/personal/gama-academy/xp-37/mini-projeto(master*) » npm run build
gama-xp-37-ts@1.0.0 build
tsc
src/1-tipos-basicos/index.ts:6:18 - error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
3 console.log(soma('1', '1'))
```

Quais problemas o TS traz?

- Necessidade de um processo de build/compilação
- Uma certa burocratização do desenvolvimento em projetos pequenos e pessoais, mas que se pagam em projetos grandes.
- Mensagens de erros nem sempre muito claras
- Falta de embasamento em JS dependendo de como é estudado

Devo sair tipando tudo?

- Muito trabalho e coisas que estamos sendo redundantes se tiparmos

O que é inferência?

- Inferir significa deduzir ou concluir algum tipo e o próprio compilador do TS é capaz de deduzir muitos tipos que permitem que o trabalho no desenvolvimento fique mais prático.
- No exemplo abaixo não deixamos explícito o tipo de retorno da função, apenas seus parâmetros, mesmo assim, o retorno é deduzido (ou inferido) pelo próprio compilador.

```
> mini-projet function soma(a: number, b: number): number
function soma (a: number, b: number) {
  return a + b;
}

console.log(soma(1, 1))
console.log(soma('1', '1'))
```

Como instalar?

```
npm init -y
```

- Começar um pacote NPM vazio
- Criamos o package.json

```
npm i --save-dev typescript OU npm i -D typescript
```

- Vamos gerar apenas no ambiente de desenvolvimento.
- Criamos o package-lock

```
npx tsc --init
```

- Executar o binário instalados no nosso projeto
- tsconfig.json arquivo criado com sucesso, são as configurações do compilador do nosso projeto.
- Temos um objeto em JSON, basicamente devemos estar atentos ao **target e module**
- O TS vai compilar todos os arquivos `.ts` do diretório
- Podemos remover comentários etc. e temos outras várias configurações à nossa necessidade

Target

- Alvo do nosso código que queremos compilar

Module

- O tipo de módulo que vamos trabalhar

src

- Onde estarão nossos arquivos TS

build

- Onde estarão nossos arquivos compilados para **JS**

```
"outDir": "./build",
/* Redirect output struc
"rootDir": "./src",
```

gitignore

- Vamos ignorar a pasta node_modules e build.

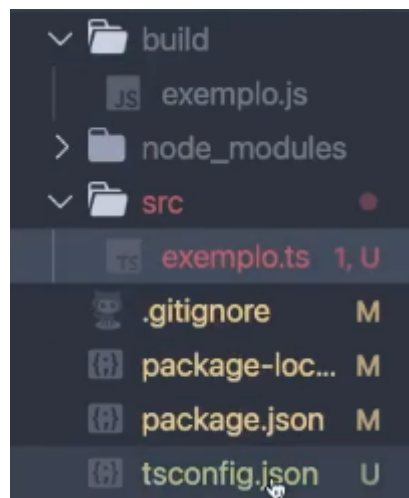
Executando o compilador

```
npx tsc
```

- Executamos nosso compilador.

Automatizando a execução do nosso compilador

- Criamos o script de build dentro do **package.json**
- `npm run build`



```
"scripts": {
  "test": "echo \"Error: no test specified\" &&
    exit 1",
  "build": "tsc"
},
```

Exemplo JS x TS

```
function soma(a, b) {
    return a + b;
}

soma(1, 1);
soma('1', '1');
```

```
function soma(a, b) {
    return a + b;
}

soma(1, 1);
soma('1', '1');
```

(parameter) a: any
Parameter 'a' implicitly has an 'any' type. ts(7006)
View Problem (⌘F8) Quick Fix... (⌘.)

Tipando nossas variáveis com TypeScript

- Erro já aparece em tela, tanto em tempo desenvolvimento quanto em tempo de build
- Enquanto no JS, ele dá o jeito e imprime o resultado, mesmo que não esperado.

```
function soma(a: number, b: number) {
    return a + b;
}

soma(1, 1);
soma('1', '1');
```

noEmitOnError

- Prevenindo a compilação com erros e não gerando o build.

```
"noEmitOnError": true,
```

Comparação com GO

- O mesmo processo de compilação acontece com o GO, onde geramos um executável para ser utilizado.
- Precisamos usar tipagem também em Go.

- `go build main.go`

Tipos básicos I

index.ts

- Criamos um `index.ts` para cada pasta dentro do `src`
- Nosso build vai ser gerado as pastas com os arquivos dentro.

String

- Vai indicar um erro sempre que tivermos outro valor a não ser string.
- Quando definimos a variável com o valor (atribuímos já na declaração), o compilador TS já identifica que ela é uma String

```
tipos basicos > string.ts > ...
let nome: string;

// ...

nome = 'gabriel';

const nomeComValor = 'gabriel';
```

```
function nomeCompleto(nome: string,
sobrenome: string) {
    return `${nome} ${sobrenome}`
}

nomeCompleto('Gabriel', 'Ramos')
```

Number

- Exemplo de tipagem com number em funções
- Garantir os tipos como numéricos e já indica erro se tivermos outro tipo não esperado.


```

let idade: number;

idade = 25;

function somaIdade(idadeInicial: number, anosSePassaram: number) {
    return idadeInicial + anosSePassaram;
}

somaIdade(25, 1)
somaIdade(['25', 1])

```

Boolean

- Valores true e false, apenas.

```

let estaAtivo: boolean;

// ..
estaAtivo = true

function mapearStatusDeUsuario(status: boolean) {
    if (status) {
        return 'Usuário está ativo'
    } else {
        return 'Usuário NÃO está ativo'
    }
}

mapearStatusDeUsuario(true)

```

Array

- Estrutura de dados com sequência de valores de forma ordenada
- Podemos definir com os próprios critérios de Array,
- Indicamos primeiro o tipo do conteúdo dentro seguido de chaves
 - `string[]` (Array de Strings)
- Só vai aceitar os valores de dentro que seguem a tipagem.

```
const gatos: string[] = [  
  'lora',  
  'logan',  
  'lebeau'  
]  
  
function exhibeGatos(gatos: string[]) {  
  return `Os gatos são: ${gatos.join(', ')}`  
}  
  
exibeGatos(gatos)
```

Para compilar utilizamos → `npm run build`

Para executar o arquivo → `node build array.js`

Tipos básicos II

Tupla

- É um array com tamanhos e tipos definidos.
- Interessante para trabalhar com valores fixos.

```
const pets: [string, string] = [  
  'lora',  
  'logan',  
]
```

Exemplo

```
const pets: [string, string, string] = [
  'lora',
  'logan',
  'lebeau'
]

const lora: [string, number] = [
  'lora', 6
]

const logan: [string, number] = [
  'logan', 5
]

const lebeau: [string, number] = [
  'lebeau', 3
]
```

Exemplo com useState

- Retorna Array com duas posições
 - É um exemplo de Tupla.
1. O valor de estado que estamos utilizando
 2. Uma função que serve para atualizar o valor da primeira variável.

→ **Em tempo de desenvolvimento estamos a validação estática do que estamos desenvolvendo**

Enum

- Forma de trabalhar com valores numéricos de forma organizada e legível
- Ele é uma representação numérico de um valor legível
- Gera um objeto em JS com números de permissões.

```
enum Permissoes {
  admin,
  editor,
  comum
}

const usuario = {
  nivel: Permissoes.admin
};
```

```
~/Development/personal/gama-academy/xp-37/typescript
/mini-projeto(master*) » node build/1-tipos-basicos/
enum.js
{ nivel: 0 }
```

- Conseguimos atribuir strings para o Enum também
- O valor gerado será uma String no Build, ao converter para JS

```
enum Permissoes {
  admin = 'ADMIN',
  editor = 'EDITOR',
  comum = 'COMUM'
}

const usuario = {
  nivel: Permissoes.admin
};

console.log(usuario)
```

- Um exemplo de uso é para paletas de cores.

```
enum Cores {
  red = '#ff0000',
  black = '#000'
}

const usuario = {
  perfil: Cores.red,
  nivel: Permissoes.admin,
};
```

Any

- É uma tipagem padrão que é atribuída para qualquer valor que não seja **tipado**.
- Evitar esse tipo de any, já que não ajuda no desenvolvimento.
- É visto como uma gambiarra, pois por padrão todos os tipos são **any**.

Unknown

- É um **any** com esteroides, mas com mais segurança que **any**.
- Se trocássemos para **any** já não teríamos erro.

```
let informacoesCompletas: string
Type 'unknown' is not assignable to type
'string'. ts(2322)
View Problem (^F8) No quick fixes available
informacoesCompletas = informacoes;
```

Null e Undefined

- Tipagens como o Null e Undefined no JS

Null

- Valor que é nulo

Undefined

- Valores para variáveis que foram criadas mas não tem valores atribuídos

Ambas não conseguimos atribuir valores, já que temos valores atribuídos.

```
let variavelNula: null;  
let variavelIndefinida: undefined;
```

Object

- Não é prático, garantimos com o **type alias** em vez deles (type)
- Não conseguimos garantir essa tipagem, pois depende das propriedades.

Com type

```
type Pessoa = {  
  name: string;  
  lastName: string;  
}  
  
let pessoa: Pessoa = {  
  name: 'gabriel',  
  lastName: 'ramos',  
}
```

```
let pessoa: object = {  
  name: 'gabriel',  
  lastName: 'ramos',  
}
```

Múltiplos tipos com Union types | type, interface

- Alguns tipos que podemos compor a partir de outros tipos
 - Alias, Intersection Type e Union
 - Retornos de função como void e never

Funções

- Toda função em JS por padrão tem um retorno, essa é a forma como o JS funciona.
- A linguagem do JS é construída com uma pilha de chamada, cada vez que a função termina, outra é executada
- Temos uma sincronia das nossas funções, pois por padrão o JS é **síncrono**
- Se o retorno não tem um valor explícito, ele volta **undefined**
- É o comportamento padrão dele.

```
function a() {

}

function b() {

}

function principal() {
  const valorA = a();
  const valorB = b();
}

principal()
```

```
function principal() {
  console.log('executando')
}

console.log(principal()) undefined
```

- Usando o TS, o compilador consegue inferir o tipo do que é retornado na função

```
mais-tipos > function principal(): number[]
function principal() {
  console.log('executando')
  return [1, 2, 4];
}

principal()
```

Void

- Tipo padrão atribuído a funções que não tem retorno, ou quando quisermos executar alguma coisa e dizer que a função não vai ter retorno explícito.

Exemplo

```
function principal(): void {  
    console.log('executando')  
}  
  
principal()
```

Never

- Relacionado ao tipo de retorno na função, mas temos dois casos
 1. Onde a função retorna um erro, a execução para até o erro ser tratado.
 2. Loop infinito dentro da função (laços de repetição infinitos), como `while=(true)`

Exemplos

```
function funcaoQueNuncaRetorna(): never {  
    while(true) {  
    }  
}  
  
funcaoQueNuncaRetorna()
```

```
function funcaoQueNuncaRetorna(): never {  
    throw new Error('ola')  
}  
  
funcaoQueNuncaRetorna()
```

Type alias

- É como trabalhamos com `type`, e por isso não usamos objetos
- Definimos a tipagem dos campos, com chave (nome do campo) e valor (tipo do dado)
- A chave e valor é campo e tipo do campo

```

type User = {
  name: string;
  lastName: string;
  birthday: string;
  age: number;
}

const gabriel: User = {
  name: 'gabriel',
  lastName: 'ramos',
  birthday: '29/01/1996',
  age: 25
}

```

Valores nulos opcionais

- Campos que podem ou não ser preenchidos com valores.
- Propriedade opcional que não será acusado erro, caso removida
- Usamos o operador `?`

```

type User = {
  name: string;
  lastName: string;
  birthday: string;
  age?: number;
}

```

Union Types

- São maracados pelo sinal de Pipe `|`
- Serve para indicar que um valor tem que ser de um tipo ou de outro aplicado pra tipagem do que esperamos
- É como se fosse o **ou**

Exemplo

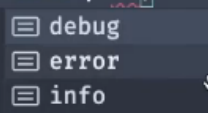

```

type LogLevel = 'info' | 'error' | 'debug';

function logMessage(message: string, level: LogLevel) {
  console.log(`[${level}] - ${message}`)
}

logMessage('Uma mensagem info', 'info')

```



Intersection types

- Combinar os **Types Alias (types)**
- São caracterizados por agrupar vários type alias, como se fosse o &.
- Acabamos ficando com dois níveis de erros, se usarmos dois types

```

type User = {
  name: string;
  lastName: string;
  birthday: string;
  age?: number;
}

```

```

// intersection types: &
type About = {
  bio: string;
  interests: string[]
}

type Profile = User & About;
const userWithProfile: Profile = {
  name: 'gabriel',
  lastName: 'ramos',
  birthday: '29/01/1996',
  bio: 'Olá, meu nome é gabriel',
  interests: ['gatos', 'música', 'fotografia']
}

```

Juntando níveis

```
type ComposedProfile = User & {  
  log: LogLevel;  
}
```

Orientação de Objetos I

Classes

- Já existe a palavra reservada Class no JS e podemos implementar uma classe.
- Mas as classes são funções no JS, que vão implementar essas classes.
- Nossa Classe é a forma como nosso Objeto vai funcionar.
- A classe é uma abstração da função

Exemplo

```
class User {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
}  
  
const pessoa = new User('gabriel', 25)  
console.log(pessoa.name) gabriel
```

- Por padrão conseguimos fazer a mesma coisa com função, o Class no JS é um Syntax Sugar.

Exemplo

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

const pessoa = new User('gabriel', 25)
console.log(pessoa)  User { name: 'gabriel', age: 25 }
```

Herança

- Estender nossa Classe que já temos
- Conseguimos herdar métodos e propriedades que temos na classe-mãe ou pai.

Exemplo

```
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

class Player extends User {
  constructor(name, age, game) {
    super(name, age);

    this.game = game;
  }
}

const jogador = new Player('Gabriel', 25, 'Ghost of Tsushima');
console.log(jogador)  Player { name: 'Gabriel', age: 25, game: 'Ghost of Tsu' }
```

Métodos Static

- Conseguimos definir métodos estáticos no JS e que não precisamos acessá-los através de um objeto.
- São métodos que podemos acessar diretamente na nossa classe, sem precisar de um objeto.

Exemplo sem Static

```
// classes e funções
class Usuario {
  public nome;
  public idade;

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

class Player extends Usuario {
  public jogo;

  constructor(nome: string, idade: number, jogo: string) {
    super(nome, idade);

    this.jogo = jogo;
  }

  dizerOJogoAtual() {
    return `Estou jogando, no momento: ${this.jogo}`;
  }
}
```

```
const jogador = new Player('Anna', 25, 'Red Dead Redemption 2');
console.log(jogador.dizerOJogoAtual())
```

Exemplo com Static

- Podemos acessar diretamente da nossa classe

```
static queHorasSao() {
  return Date();
}
```

```
const jogador = new Player('Gabriel', 25, 'Ghost of Tsushima');
console.log(jogador.dizerOJogoAtual())
console.log(Player.queHorasSao())
```

Orientação de Objetos II

Diferenças entre método `public`, `private` e `protected`

- Forma como podemos acessar alguma classe.
- Um objeto é a instância de uma classe

Public

- Podemos acessar a propriedade do objeto de forma geral, dentro e fora da classe

```
class Jogo {  
    public nome;  
  
    constructor(nome: string) {  
        this.nome = nome;  
    }  
}  
  
const ghost = new Jogo('Ghost of Tsushima');  
console.log(ghost.nome);
```

Privado

- Não deixamos o valor da propriedade acessível para todos.
- Só deixamos acessível dentro da classe.
- Conseguimos o nome através de um método (normalmente Setter)

```
class Jogo {  
    private nome;  
  
    constructor(nome: string) {  
        this.nome = nome;  
    }  
  
    dizerNome() {  
        return `O nome do jogo é: ${this.nome}`;  
    }  
}  
  
const ghost = new Jogo('Ghost of Tsushima');  
console.log(ghost.dizerNome());
```

Protected

- Não temos o nome dentro da classe, quando colocamos privado.
- Só conseguimos acessar propriedade de classes com herança através do protected.

- Quando usamos extends (herança), só conseguimos acessar propriedades dela com protected

Interfaces (e semelhanças com type alias)

- São semelhante com `Type` e formas de garantir campos e métodos de funções das classes, ou tipos de dados.
- Podemos colocar o `INomeDaInterface`, sempre com o `I` na frente ou não.
- Ela funciona como um contrato.
- Definimos de forma pública os campos.

```
interface IJogoComDescricao {
  nome: string;
  descricao: string;
  dizerNomeComDescricao(): string;
}

class JogoComDescricao extends Jogo implements IJogoComDescricao {
```

- Conseguimos também implementar para objetos, assim como o `Type`

```
const obj: IJogoComDescricao = {
  descricao: 'descricao do jogo',
  dizerNomeComDescricao() {
    return '123'
  }
}
```

Usar Type ou Interface

- Depende do projeto, cada um segue um padrão
- Quando redefinimos `Interface`, mesclamos com a Interface definida anteriormente, somando as tipagens feitas.
- Com `Type` usamos o operador `&&`

Type Assertion

- Indica ao compilador do TS para garantir e confiar que a tipagem que estamos marcando é de outro tipo específico.

Fazendo com `As`

```
type JogoAssertion = {  
  nome: string;  
  descricao: string;  
}  
  
let jogo = {} as JogoAssertion;  
jogo.nome = 'ausihdahusd'
```

Usando generics <> ★

- É a mais usada no TypeScript com React.

```
type JogoAssertion = {  
  nome: string;  
  descricao: string;  
}  
  
let jogo = <JogoAssertion>{};  
jogo.nome = 'nome'  
jogo.descricao = 'descricao do jogo'
```