

FIAP

FIAP

SLIDER ▢■◀





TypeScript



O que é TypeScript

- O TypeScript foi criado pela Microsoft, lançado em 2012, para resolver problemas associados ao desenvolvimento de grandes aplicativos JavaScript, proporcionando uma linguagem com tipagem estática que melhora a manutenção, escalabilidade, e a experiência do desenvolvedor.
- O TypeScript surge como um superconjunto do JavaScript, introduzindo recursos poderosos como a **tipagem estática**. Isso significa que, em vez de adivinhar os tipos de dados das suas variáveis e funções, você poderá defini-los explicitamente, garantindo um código mais preciso e livre de erros.
- Documentação oficial: <https://www.typescriptlang.org/docs>

TypeScript x Javascript

Vamos construir uma função que calcula uma área determinada, que a formula é $\text{base} * \text{altura}$ com Javascript:

```
1  function calculateArea(base, height) {  
2    |   return base * height  
3  }  
4  
5  console.log(calculateArea(2, 3)) // 6  
6  
7  
8
```

Console ×

6

TypeScript x Javascript

A função parece funcionar perfeitamente quando passamos valores numéricos. Mas o Javascript permite que eu passe uma String como argumento da minha função `calculateArea`. Vamos testar passar o número 2 como string e ver como ele se comporta

TypeScript x Javascript

A função parece funcionar perfeitamente quando passamos valores numéricos. Mas o Javascript permite que eu passe uma String como argumento da minha função `calculateArea`. Vamos testar passar o número 2 como string e ver como ele se comporta

script.js ×

```
1 function calculateArea(base, height) {  
2   |   return base * height  
3 }  
4  
5 console.log(calculateArea('2', 3)) // 6  
6
```

Console ×

6

Aparentemente tudo continua funcionando muito bem. Por debaixo dos panos o JavaScript entendeu que a string poderia ser convertida para um número, fez a conversão automática e depois fez a operação.

TypeScript x Javascript

Agora vamos estressar um pouco mais o problema. O que vai acontecer se eu passar o valor da base como a string 'Oi'?

TypeScript x Javascript

Agora vamos estressar um pouco mais o problema. O que vai acontecer se eu passar o valor da base como a string 'Oi'?

script.js ×

```
1  function calculateArea(base, height) {  
2    |   return base * height  
3  }  
4  
5  console.log(calculateArea('Oi', 3))  
6
```

Console ×

nan

Agora tivemos um problema. O resultado de saída foi o `nan` (ou NaN, que é o identificador do Javascript para Not A Number). Ou seja, não foi uma operação matemática válida.

TypeScript x Javascript

Normalmente para corrigirmos esse tipo de situação, costumamos fazer um tratamento de erros na entrada na função para que não seja permitido números inválidos.

```
1  function calculateArea(base, height) {  
2      if (typeof base !== 'number' || typeof height !== 'number') {  
3          throw new Error('base and height must be numbers')  
4      }  
5      return base * height  
6  }  
7  
8  console.log(calculateArea(2, 3)) // 6  
9  
10 console.log(calculateArea('0i', 3)) // Error: base and height must be numbers
```

Com o nosso if validando se as entradas são numéricas, agora estamos protegidos de um resultado não esperado. Tudo isso com Javascript puro.

TypeScript x Javascript

Em TypeScript conseguimos fazer a mesma coisa de uma forma muito mais simples: Atribuindo tipos aos valores de entrada

```
function calculateAreaTypescript(base: number, height: number) {  
  return base * height  
}
```

```
console.log(calculateAreaTypescript(2, 3)) // 6
```

BugFinder: Argument of type 'string' is not assignable to parameter of type 'number'.

```
console.log(calculateAreaTypescript('Oi', 3))
```

Tipamos a base como numero e a altura também. E percebam que o próprio editor de texto já me avisou que temos um erro. Diferente quando estávamos escrevendo com JavaScript puro. Isso significa que vamos identificar o erro em **Tempo de Desenvolvimento**, antes de mandar a aplicação para um ambiente produtivo, já que o compilador não vai deixar enviar o código com erros.

Sintaxe da Tipagem

Como vimos no exemplo anterior a sintaxe é bastante simples.

Para tiparmos uma variável, um parâmetro ou a saída de uma função, basta utilizamos o caractere de : (dois pontos) seguido do tipo a ser especificado.

1. Tipando uma variável

```
1  const text: string = 'Olá!'
```

2. Tipando a entrada de uma função

```
3  function testing(input: string): string {  
4  |    return input  
5  |  }
```

3. Tipando a saída de uma função

```
3  function testing(input: string): string {  
4  |    return input  
5  |  }
```

Tipos primitivos e Tipos complexos

Como vimos no exemplo anterior a sintaxe é bastante simples. Podemos usar qualquer um dos tipos primitivos do Javascript:

- **number** representa números inteiros e decimais (ex: 1, 3.14, -50).
- **string** representa sequências de caracteres (ex: "Olá, mundo!", 'Uma frase com aspas simples').
- **boolean** representa valores lógicos (ex: true, false).
- **null** representa a ausência de valor.
- **undefined** representa um valor não inicializado.

Tipos primitivos e Tipos complexos

Além disso temos o tipo **void**, que representa a ausência de valor. Ele é frequentemente utilizado em contextos onde uma função não retorna nenhum valor concreto, apenas executa uma ação ou modifica o estado do programa. Imagine um cenário em que você deseja imprimir uma mensagem no console, mas não precisa armazenar o retorno dessa operação em nenhuma variável. É aí que o **void** entra em cena

```
function imprimirMensagem(mensagem: string): void {  
    console.log(mensagem);  
}  
  
imprimirMensagem("Olá, mundo!"); // A função não retorna valor
```


Tipos primitivos e Tipos complexos

Além do void, temos o tipo **unknown** (desconhecido), que representa um valor cujo não tenhamos conhecimento do que virá. Imagine que você está recebendo dados de uma fonte externa, como uma API ou um arquivo JSON, e não sabe o tipo exato desses dados. É aí que o tipo desconhecido entra em cena:

```
function obterDadosAPI(url: string): unknown {  
    // Simula a recuperação de dados de uma API  
    return { nome: "João Silva", idade: 30 };  
}  
  
const dadosAPI: unknown = obterDadosAPI("https://exemplo.com/api");
```

Tipos primitivos e Tipos complexos

Também temos o tipo **any** (qualquer) que serve como um tipo coringa. Literalmente ele aceita qualquer coisa, e se algo é declarado sem um tipo especificado, automaticamente o Typescript entende que é do any. Esse tipo deve ser evitado, pois é uma má pratica e praticamente anula o uso e o ganho que temos com Typescript, salvo raras exceções.

```
1  let test: any = 'a'
2
3  test = 1
4
5  test = true
6
7  let test2: string = 'a'
8
9
10
11
12
13 test2 = 1
```

Type 'number' is not assignable to type 'string'. (2322)

let test2: string

[View Problem \(Alt+F8\)](#) No quick fixes available

Tipos primitivos e Tipos complexos

Os tipos complexos representam estruturas de dados mais elaboradas que podem conter diversos valores e funcionalidades. No TypeScript, os tipos complexos mais comuns são:

- **Objetos** Coleções de pares chave-valor que armazenam dados relacionados.
- **Arrays** Listas ordenadas de valores do mesmo tipo.
- **Funções** Blocos de código reutilizáveis que executam tarefas específicas.
- **Tuplas** Listas fixas de valores com tipos específicos.
- **Enum** Conjuntos de valores nomeados e constantes.

```
1 // Objeto representando um usuário
2 let user: { name: string, age: number } = { name: "João Silva", age: 30 };
3
4 // Array de números
5 let notes: number[] = [10, 8, 7, 9, 5];
6
7 // Função que retorna a soma de dois números
8 function sum(x: number, y: number): number {
9   | return x + y;
10  }
11
12 // Tupla representando um ponto no espaço
13 let dot: [number, number] = [10, 20];
14
15 // Enum representando os dias da semana
16 enum WeekDay { Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado };
17 let weekDay: WeekDay = WeekDay.Terca;
18
```

Testando o TSC (TypeScript Compiler)

Agora vamos testar na prática.

1. Crie uma pasta e inicie o projeto com o comando `npm init` que já aprendemos anteriormente
2. Vamos instalar o TypeScript com o comando `npm install typescript` (também podemos instalar globalmente com a flag `-g` no final comando)
3. Agora vamos rodar o comando de execução de script do npm, o npx: `npx tsc --init`

```
(base) PS C:\Users\vinix\Desktop\Projeto1> npx tsc --init
```

Created a new tsconfig.json with:

```
target: es2016
module: commonjs
strict: true
esModuleInterop: true
skipLibCheck: true
forceConsistentCasingInFileNames: true
```

You can learn more at <https://aka.ms/tsconfig>

Testando o TSC (TypeScript Compiler)

Agora vamos testar na prática.

4. O comando vai gerar um arquivo **tsconfig.json** com diversas configurações do TypeScript comentadas e algumas já funcionais. Não vamos passar por elas agora, mas recomendo a leitura sobre o arquivo de configuração <https://aka.ms/tsconfig>

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig to read more about this file */

    /* Projects */
    // "incremental": true,           /* Save .tsbuildinfo files to allow for incremental compilation of projects. */
    // "composite": true,           /* Enable constraints that allow a TypeScript project to be used with project references. */
    // "tsbuildinfoFile": "./.tsbuildinfo", /* Specify the path to .tsbuildinfo incremental compilation file. */
    // "disableSourceOfProjectReferenceRedirect": true, /* Disable preferring source files instead of declaration files when referencing composite projects. */
    // "disableSolutionSearching": true, /* Opt a project out of multi-project reference checking when editing. */
    // "disableReferencedProjectLoad": true, /* Reduce the number of projects loaded automatically by TypeScript. */

    /* Language and Environment */
    "target": "es2016", /* Set the JavaScript language version for emitted JavaScript and include compatible library declarations. */
    // "lib": [],           /* Specify a set of bundled library declaration files that describe the target runtime environment. */
    // "jsx": "preserve",   /* Specify what JSX code is generated. */
    // "experimentalDecorators": true, /* Enable experimental support for legacy experimental decorators. */
    // "emitDecoratorMetadata": true, /* Emit design-type metadata for decorated declarations in source files. */
    // "jsxFactory": "",     /* Specify the JSX factory function used when targeting React JSX emit, e.g. 'React.createElement' or 'h'. */
    // "jsxFragmentFactory": "", /* Specify the JSX fragment reference used for fragments when targeting React JSX emit e.g. 'React.Fragment' or 'Fragment'. */
    // "jsxImportSource": "", /* Specify module specifier used to import the JSX factory functions when using 'jsx: react-jsx*'. */
    // "reactNamespace": "", /* Specify the object invoked for 'createElement'. This only applies when targeting 'react' JSX emit. */
    // "noLib": true,        /* Disable including any library files, including the default lib.d.ts. */
    // "useDefineForClassFields": true, /* Emit ECMAScript-standard-compliant class fields. */
    // "moduleDetection": "auto", /* Control what method is used to detect module-format JS files. */

    /* Modules */
    "module": "commonjs", /* Specify what module code is generated. */
    // "rootDir": "./",     /* Specify the root folder within your source files. */
    // "moduleResolution": "node16", /* Specify how TypeScript looks up a file from a given module specifier. */
    // "baseUrl": "./",     /* Specify the base directory to resolve non-relative module names. */
    // "paths": [],          /* Specify a set of entries that re-map imports to additional lookup locations. */
    // "rootDirs": [],       /* Allow multiple folders to be treated as one when resolving modules. */
    // "typeRoots": [],      /* Specify multiple folders that act like './node_modules/@types'. */
    // "types": [],           /* Specify type package names to be included without being referenced in a source file. */
    // "allowSyntheticDefaultImports": true, /* Allow accessing UMD globals from modules. */
    // "moduleSuffixes": [], /* List of file name suffixes to search when resolving a module. */
    // "allowImportingTsExtensions": true, /* Allow projects to include TypeScript file extensions. Requires 'moduleResolution bundler' and either 'noEmit' or 'emitDeclarationOnly' to be set. */
    // "resolvePackageJsonExports": true, /* Use the package.json 'exports' field when resolving package imports. */
    // "resolvePackageJsonImports": true, /* Use the package.json 'imports' field when resolving imports. */
    // "customConditions": [], /* Conditions to set in addition to the resolver-specific defaults when resolving imports. */
    // "resolveJsonModule": true, /* Enable importing .json files. */
    // "allowArbitraryExtensions": true, /* Enable importing files with any extension, provided a declaration file is present. */
    // "noResolve": true,    /* Disallow 'import', 'require's or <reference>'s from expanding the number of files TypeScript should add to a project. */
  }
}
```


Testando o TSC (TypeScript Compiler)

Agora vamos testar na prática.

5. Vamos criar um arquivo `index.ts` (ts é a extensão dos arquivos de TypeScript padrão)
6. Dentro do arquivo vamos colocar a função de calculo de área que eu mostrei anteriormente

```
1 function calculateAreaTypescript(base: number, height: number): number {  
2   |   return base * height  
3 }  
4  
5 console.log(calculateAreaTypescript(2, 3)) // 6
```

Testando o TSC (TypeScript Compiler)

Agora vamos testar na prática.

7. Agora vamos rodar o comando `npx tsc`

8. Se o nosso código não tiver erros, o compilador vai gerar um arquivo `index.js` (de javascript puro) com o código de saída.

```
    "use strict";  
    ✓ function calculateAreaTypescript(base, height) {  
      |   return base * height;  
    }  
    console.log(calculateAreaTypescript(2, 3)); // 6  
    .
```

Testando o TSC (TypeScript Compiler)

Podemos notar que o TypeScript apenas adicionou uma string “**use strict**” no começo do nosso código. Para casos mais complexos a saída costuma ser mais complexa, mas apenas para fins didáticos o nosso TypeScript está funcionando.

O **use strict** é uma diretiva que foi introduzida no ECMAScript 5 (ES5) para permitir um modo de execução "estrito" que introduz mudanças na semântica do JavaScript.

- Elimina alguns erros silenciosos do JavaScript, fazendo com que eles gerem exceções.
- Impede que determinadas ações sejam realizadas, como o uso de variáveis não declaradas.
- Evita certos usos do `this` que são considerados confusos ou não desejados.

Testando o TSC (TypeScript Compiler)

E se o meu código estiver com erro na tipagem, o que acontece?

Voltamos ao exemplo anterior onde passamos uma string “Olá” como parâmetro da nossa função, mesmo que o editor já funcione, vamos rodar de novo o comando `npx tsc` e ver a saída no nosso terminal, que deve ser um erro de tipagem

```
index.ts > ...
```

```
1  function calculateAreaTypescript(base: number, height: number): number {  
2    |   return base * height  
3  }  
4  
5  console.log(calculateAreaTypescript(2, 3)) // 6  
6  
7  console.log(calculateAreaTypescript('Ola', 3))  
8
```

```
(base) PS C:\Users\vinix\Desktop\Projeto1> npx tsc
```

```
index.ts:7:37 - error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
```


```
7  console.log(calculateAreaTypescript('Ola', 3))  
                                     ~~~~~
```

```
Found 1 error in index.ts:7
```

Testando o TSC (TypeScript Compiler)

Porém o meu index.js ainda sim foi gerado. Mesmo com o terminal dando erro. Isso é um comportamento padrão de configuração do TypeScript. Mas vamos mudar isso: Vamos ao arquivo **tsconfig.json** e “ligar” a propriedade **noEmitOnError**

```
tsconfig.json / tsconfigOptions
{
  "compilerOptions": {
    // "emitComments": true, /* Disable emitting comments. */
    // "noEmit": true, /* Disable emitting files from a compilation. */
    // "importHelpers": true, /* Allow importing helper functions from tslib once per project, instead of including them per-file. */
    // "downlevelIteration": true, /* Emit more compliant, but verbose and less performant JavaScript for iteration. */
    // "sourceRoot": "", /* Specify the root path for debuggers to find the reference source code. */
    // "mapRoot": "", /* Specify the location where debugger should locate map files instead of generated locations. */
    // "inlineSources": true, /* Include source code in the sourcemaps inside the emitted JavaScript. */
    // "emitBOM": true, /* Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files. */
    // "newLine": "crlf", /* Set the newline character for emitting files. */
    // "stripInternal": true, /* Disable emitting declarations that have '@internal' in their JSDoc comments. */
    // "noEmitHelpers": true, /* Disable generating custom helper functions like '__extends' in compiled output. */
    // "noEmitOnError": true, /* Disable emitting files if any type checking errors are reported. */
    // "preserveConstEnums": true, /* Disable erasing 'const enum' declarations in generated code. */
    // "declarationDir": ".", /* Specify the output directory for generated declaration files. */
  }
}
```



```
tsconfig.json / tsconfigOptions
{
  "compilerOptions": {
    // "emitComments": true, /* Disable emitting comments. */
    // "noEmit": true, /* Disable emitting files from a compilation. */
    // "importHelpers": true, /* Allow importing helper functions from tslib once per project, instead of including them per-file. */
    // "downlevelIteration": true, /* Emit more compliant, but verbose and less performant JavaScript for iteration. */
    // "sourceRoot": "", /* Specify the root path for debuggers to find the reference source code. */
    // "mapRoot": "", /* Specify the location where debugger should locate map files instead of generated locations. */
    // "inlineSources": true, /* Include source code in the sourcemaps inside the emitted JavaScript. */
    // "emitBOM": true, /* Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files. */
    // "newLine": "crlf", /* Set the newline character for emitting files. */
    // "stripInternal": true, /* Disable emitting declarations that have '@internal' in their JSDoc comments. */
    // "noEmitHelpers": true, /* Disable generating custom helper functions like '__extends' in compiled output. */
    "noEmitOnError": true, /* Disable emitting files if any type checking errors are reported. */
  }
}
```


Testando o TSC (TypeScript Compiler)

Agora vamos apagar o index.js e rodar o comando `npx tsc` novamente com o erro no nosso código. O resultado esperado é recebermos o mesmo erro no nosso terminal que anteriormente, porém dessa vez não pode gerar um novo index.js, porque falamos para o compilador através da flag `noEmitOnError` para não gerar nada em caso de erro de compilação.

Dúvidas, críticas ou sugestões?

FIAP