Typescript

Tipos primitivos

Mais utilizaveis!

```
Strings:
    const name: string = 'Gabriel';

Numbers:
    const yearsOld: number = 22;

Boolean:
    const isTrue: boolean = true;
```

O correto é sempre usar a tipagem em minúsculo (RECOMENDADO PELO TS)

Tipagem em arrays

```
let friendList: string[] = ['John', 'Kevin', 'Curry', 'Lady Bug'];

let commonNumbers: number[] = [1,2,3,4,5,6];

Ou voce pode usar o Generic Type:

let friendList: Array<string> = ['John', 'Kevin', 'Curry', 'Lady Bug'];

let commonNumbers: Array<number> = [1,2,3,4,5,6];
```

Tipo "any", quando usar?

```
any = qualquer coisa.
Você está mandando o TS ignorar a validação de tipos nessa variavel.

let nums: any = [1,2,3,4,5,6,7];
nums.push('Gabriel');
```

Tipos nos parâmetros de funções

```
const firstLetterUpperCase = (name: string) => {
    let fLetter = name.charAt(0).toUpperCase();
    return `${fLetter+name.substring(1)}`;
};

firstLetterUpperCase('gabriel'); // return Gabriel
```

Tipos no retorno

```
const firstLetterUpperCase = (name: string): string => {
    let fLetter = name.charAt(0).toUpperCase();
    return `${fLetter+name.substring(1)}`;
}

const yearsOld = (age: number): number => {
    const actualYear: number = new Date().getFullYear();
    const yO: number = (actualYear-age);
    return yO;
}

yearsOld(1999);
```

Contextual Typing

Temos a seguinte váriavel com um método forEach:

```
let friendList = ['Gabriel', 'John', 'Markson'];

friendList.forEach(friend => {
    console.log(friend.toUpperCase());
});
```

Mesmo não tipando meu array o Typescript usa o conceito de Contextual Typing, ele é inteligente o suficiente para olhar para o meu array, detectar que existem somente strings e quando eu atribui o método toUpperCase (destinado a strings) ele não da erro. Caso eu tivesse algum outro dado com tipagem diferente (number, boolean) ele iria me acionar sobre o erro.

Tipos em objetos (MUITO UTILIZÁVEL)

Vamos partir da situação que por algum acaso você necessite receber os dados do usuário na sua função.

Você deverá passar o parâmetro user e dizer quais propriedades do objeto quer receber e quais seus tipos, independente se o usuário enviar mais campos, você está apenas utilizando os que você esta declarando no parâmetro da função.

```
const getDataUser = (user: {name: string, years: number}) => {
    console.log(`Hello ${user.name}, you have ${user.years} years`);
};

const user = {
    name: 'Gabriel',
    years: 22
}

getDataUser(user);
```

Propriedades Opcionais

Utilizando a mesma situação acima, mas digamos que demos a opção do usuário de querer ou não enviar sua idade, como fazemos isso? Simples, adicionamos o símbolo "?" antes dos dois ponto do parâmetro.

```
const getDataUser = (user: {name: string, years?: number}) => {
    console.log(`Hello ${user.name}, you have ${user.years} years`);
};
```

Union Types

Váriaveis com múltiplos tipos.

Recebendo o dado numa váriavel e exibindo em um HTML num componente qualquer.

Você está especificando que essa sua váriavel pode receber esses dois tipos (**Utilize em casos bem específicos**)

```
let name: string | number = 22;
name = document.getElementById('username').innerHTML;
```

Types e Interface

Criação de próprios tipos: Normalmente para criação de tipos é utilizavel a atribuição de nome do tipo Paschal Case (MeuTipo).

```
type Years = number;
let yO: Years = 22;

const showYearsOld = (y: Years): Years => {
    console.log(y);
}
```

A utilização de criação de tipos fica bem mais nítida quando estamos trabalhando da maneira a seguir... Lembra dessa função?

```
const getDataUser = (user: {name: string, years?: number}) => {
    console.log(`Hello ${user.name}, you have ${user.years} years`);
};
```

Se o seu usuário tivesse 30 parâmetros você iria ter que toda vez passar os 30 parâmetros na função, então para isso, você poderá criar o seu type User:

```
type User = {name: string, years: number};

const getDataUser = (user: User) => {
    console.log(`Hello ${user.name}, you have ${user.years} years`);
};
```

Existe outra maneira de criar tipos, chamada Interface.

```
interface User {name: string, years: number};
```

**Os 2 fazem a mesma coisa, porém com o type não é possível editar seu tipo ao longo do código, já com interface eu poderia declarar 2x, exemplo:

```
interface User {name: string};
interface User {years: number};
```

Type Assertions

É uma maneira de você "ajudar" o Typescript a entender melhor o que você está criando, vamos lá... Supondo que você esteja pegando informações do seu form em HTML:

```
const submitFormUser = document.getElementById('user-submit');

console.log(submitFormUser.value);
```

Você irá notar que o "value" ficará sublinhado de vermelho, isso **NÃO** é um erro, porque em Javascript esse código está certo, sua váriavel que é um HTMLElement possui o método value. Más o TS não entende isso e aponta para você e ai que entra o Type Assertions, você irá especificar que aquele HTMLElement é do tipo input, dessa maneira:

```
const submitFormUser = document.getElementById('user-submit') as HTMLInputElement;

console.log(submitFormUser.value);
```

Tipos literais

Você pode definir os reais valores que sua váriavel pode ter! Pera, como assim? Digamos que no seu site você possibilite o usuário digitar em que lado era irá alinhar o texto (right, left ou center), mas ele pode simplesmente digitar "léfiti" que não é um alinhamento, então você pode usar Type Literals para isso.

```
const getUserTextAlign = (text: string, alignment: 'left' | 'right' | 'center') => {
    return `<div> style="text-align:${alignment}">${text}</div>`;
};
```

Tipos para funções

Vamos criar um tipo para uma função matemática que vai receber 2 numbers e fazer uma conta de nível básico.

```
type MathType = (n1: number, n2: number) => number;

const sumMath: MathType = (n1,n2) => { return n1 + n2; };
const mulMath: MathType = (n1,n2) => { return n1 * n2; };
const divMath: MathType = (n1,n2) => { return n1 / n2; };
const subMath: MathType = (n1,n2) => { return n1 - n2; };
```

Retorno vazio (type void)

A função vai ser executada mas ela não te retorna nada.

```
const funcVoid = () => void;
```