



eFront - TypeScript

Introdução

Como já dito, a tipagem dinâmica nos permite atribuir diferentes tipos valores para uma variável, hora pode receber uma string, hora um number e isso pode prejudicar a nossa aplicação a medida que ela vai crescendo. Exemplo: imagina que você tem uma variável chamada “nome” que no decorrer da aplicação ela recebe um número, isso é aceitável no JavaScript, mas não é nossa intenção ao criar variável “nome”.

```
1 let nome = "iuri";  
2 nome = 12;
```

Porém, isso já não acontece em linguagens que possuem tipagem estática, pois, nessa linguagens obrigatoriamente precisamos declarar o tipo do dado que a variável irá receber.

```
1 let nome:string = "iuri";  
2 nome = 12; // Teremos um erro
```

A mensagem de erros será: O tipo 'number' não pode ser atribuído ao tipo 'string'.

Introdução a TypeScript: o que é e para que serve

Pensando nesse problema a Microsoft desenvolveu o TypeScript para tiparmos os dados, porém de uma forma que mantém toda a sintaxe e funcionamento do JavaScript que já conhecemos. Mas acaba entrando uma questão: "Mas luri, o navegador não entende só JavaScript?". Exato! No final de todo o processo, o código que desenvolvemos utilizando TypeScript irá virar JavaScript comum, isso funciona da mesma maneira do Sass.

O que exatamente o TypeScript faz?

O TypeScript detecta quando passamos um valor diferente para uma variável declarada durante o desenvolvimento e avisa na IDE (no seu editor de código). Isso reflete num ambiente muito mais seguro enquanto o código está sendo digitado. Outra coisa interessante do TypeScript é a possibilidade de transpilar o seu código para uma versão antiga do JavaScript, isso faz com que seu código seja lido por todas as versões dos navegadores.

Instalando o TypeScript

Para utilizarmos o TypeScript precisamos do Node.js instalado. Depois do Node, começaremos instalando o TypeScript globalmente em nossa máquina. Para isso daremos o seguinte comando:

```
npm install -g typescript
```

Agora com o TypeScript instalado, podemos utilizar o comando "tsc" no nosso terminal. Onde usaremos o tsc para executar o nosso arquivo chamado "app.ts". Obs: para um arquivo rode código TypeScript, ele precisa ter a extensão .ts

```
tsc app.ts
```

Perceba que agora um arquivo app.js foi criado na pasta raiz do nosso projeto.

Isso aconteceu porque o tsc transformou toda linguagem TypeScript em JavaScript.

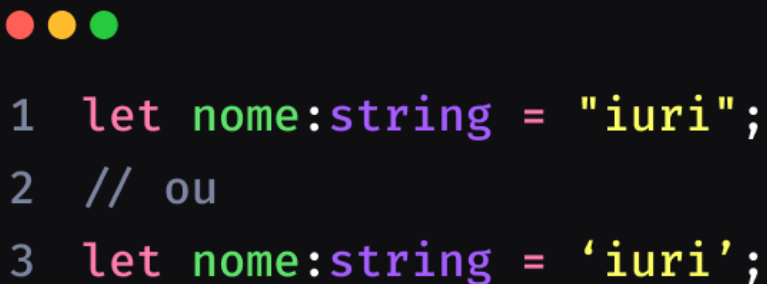
Caso tivéssemos algum recurso que só existe no TypeScript, ele também seria traduzido de forma que o node pudesse entender.

Tipos em TypeScript

Tipos são valores que informamos como um complemento de uma variável, a fim de evitar que seja atribuído um valor diferente de variável.

String

String é um tipo de dados usado para armazenar dados de texto. Os valores de string são colocados entre aspas simples ou aspas duplas.



```
1 let nome:string = "iuri";  
2 // ou  
3 let nome:string = 'iuri';
```

Template String

Template Strings são usados para incorporar expressões em strings.

Aqui, em vez de escrever uma string que é uma combinação de texto e variáveis com concatenações, podemos usar uma única instrução com back-ticks/crase(`). Os valores das variáveis são escritos como \${}.



```
1 let nome:string = "iuri";  
2 let frase:string = `${nome} é dev`;
```

Boolean

Os valores booleanos são utilizados para armazenar valores true ou false.



```
1 let dev:boolean = true;
```

Number

Todos os números são armazenados como números de ponto flutuante. Esses números podem ser decimais (base 10), hexadecimal (base 16) ou octal (base 8).



```
1 let idade:number = 22;
```

Array

Um array pode armazenar vários valores de diferentes tipos de dados sequencialmente usando uma sintaxe especial. Existem duas maneiras de declarar um array:

1. Usando colchetes. Este método é semelhante a como você declararia array em JavaScript.



```
1 let nomes: string[] = ['Iuri', 'Eiji'];
```


2. Usando um tipo de array genérico, Array <elementType>.



```
1 let nomes: Array<string> = [  
2     'Iuri', 'Eiji'  
3 ];
```

Ambos os métodos produzem a mesma saída.


Um array pode conter elementos de diferentes tipos de dados, para isso no TypeScript usamos uma sintaxe de tipo de array genérico, conforme mostrado abaixo.



```
1 let nomes: (string | number)[] = [  
2   'Iuri', 22, 'Eiji', 21  
3 ];  
4  
5 // ou  
6  
7 let nomes: Array<string | number> = [  
8   'Iuri', 22, 'Eiji', 21  
9 ];
```

Tuple

O TypeScript introduziu um novo tipo de dados chamado Tuple. O tuple pode conter dois valores de diferentes tipos de dados, assim, eliminando a necessidade de declarar duas variáveis diferentes.



```
1 let empresa: [number, string] = [  
2   2022, 'Google'  
3 ];
```

Enum

Em palavras simples, enums nos permite declarar um conjunto de constantes nomeadas, ou seja, uma coleção de valores relacionados que podem ser valores numéricos ou strings. Enums são definidos usando a palavra-chave enum.

Existem três tipos de enums:

1. Enum numérico
2. Enum string
3. Enum heterogêneo

Enum numérico

Enums numéricos são enums baseados em números, ou seja, armazenam valores de string como números.

No exemplo abaixo, temos um enum chamado Direcao. O enum tem quatro valores: Norte, Leste, Sul e Oeste.



```
1 enum Direcao {  
2     Norte,  
3     Leste,  
4     Sul,  
5     Oeste,  
6 }
```

Os Enums são sempre atribuídos a valores numéricos quando são armazenados. O primeiro valor sempre assume o valor numérico de 0, enquanto os outros valores no enum são incrementados em 1.

```
1 enum Direcao {
2     Norte, // 0
3     Leste, // 1
4     Sul, // 2
5     Oeste, // 3
6 }
```

Enum string

Enums de string são semelhantes aos enums numéricos, exceto que os valores de enum são inicializados com valores de string em vez de valores numéricos.

Os benefícios de usar enums de string é que enums de string oferecem melhor legibilidade. Se tivéssemos que depurar um programa, seria mais fácil ler valores de string do que valores numéricos.

```
1 enum Direcao {
2     Norte = "Norte",
3     Leste = "Leste",
4     Sul = "Sul",
5     Oeste = "Oeste"
6 }
```


A diferença entre enums numéricos e de string é que os valores de enum numéricos são incrementados automaticamente, enquanto os valores de enum de string precisam ser inicializados individualmente.

Enum heterogêneo

Enums heterogêneos são enums que contêm valores de string e numéricos.

```
1 enum Direcao {  
2     Norte = 'OK',  
3     Leste = 1,  
4     Sul,  
5     Oeste,  
6 }
```

Any

Nem sempre temos conhecimento prévio sobre o tipo de algumas variáveis, especialmente quando existem valores inseridos pelo usuário em bibliotecas de terceiros. Nesses casos, precisamos de algo que possa lidar com conteúdo dinâmico. O tipo Any é útil aqui.

```
1 let nome:any = "iuri";  
2 nome = 12; // Não teremos um erro
```

Da mesma forma, você pode criar um array do tipo any [] se não tiver certeza sobre os tipos de valores que podem conter essa array.

Atenção: A utilização do tipo any não é recomendado! Pois assim irá perder a magia do TypeScript.

Void

O void é usado onde não há dados. Por exemplo, se uma função não retornar nenhum valor, você pode especificar void como tipo de retorno.

```
1 function mensagem():void {  
2   console.log('iuricode');  
3 }
```

Never

O tipo never é usado quando você tem certeza de que algo nunca acontecerá. Por exemplo, você escreve uma função que não retorna ao seu ponto final.

```
1 function repeticao():never {  
2   while(true) {  
3     console.log('iuricode');  
4   }  
5 }
```

No exemplo acima, a função repeticao() está sempre em execução e nunca atinge um ponto final porque o loop while nunca termina. Assim, o tipo never é usado para indicar o valor que nunca ocorrerá ou retornará de uma função.

Union

No exemplo abaixo, a variável “dev” é do tipo union. Portanto, você pode passar um valor de string ou um valor numérico. Se você passar qualquer outro tipo de valor, por exemplo, booleano, o compilador dará um erro.

```
1 let dev: (string | number);  
2  
3 dev = 123;  
4 dev = 'iuri';  
5 dev = true; // Teremos um erro
```

Interface no TypeScript

No TypeScript, temos as interfaces que são um conjunto de métodos e propriedades que descrevem um objeto. As interfaces desempenham a função de nomear esses tipos e são uma maneira poderosa de definir “contrato”. Isso significa que uma estrutura que implementa uma interface é obrigada a implementar todos os seus dados tipados.

Primeiro vamos criar um exemplo sem o uso de uma interface. E entender a necessidade do uso de uma interface:

```
1 function somar(x, y) {  
2   return x + y;  
3 }
```

Neste exemplo não garantimos qual é o tipo que a propriedade x e y irá receber (isso é, uma string, number, boolean..). Já com a interface, garantimos qual será o tipo da nossa propriedade. Com ela, criamos um “contrato”, assim sabemos exatamente o que deve ser passado como parâmetro para as propriedades, e quem recebe o parâmetro também saberá.

Definindo uma Interface com TypeScript


Iremos declarar uma interface com a palavra-chave Pessoa. As propriedades e métodos que compõem a interface são adicionados dentro das chaves como key:value pares.

```
1 interface Pessoa {  
2   nome: string;  
3   idade: number;  
4 }
```

Repare que as propriedades da interface devem ser separadas por “;”.

Criando um objeto baseado numa Interface

Depois de criado a interface Pessoa, podemos criar um objeto onde iremos passar o valor para as propriedades da interface.



```
1 interface Pessoa {  
2   nome: string;  
3   idade: number;  
4 }  
5  
6 let dev: Pessoa = {  
7   nome: 'Iuri',  
8   idade: 22,  
9 }
```

Propriedades opcionais

Nem sempre temos certeza que um dado será exibido ou lido em nossa aplicação, para isso temos a propriedade opcional do TypeScript, que são marcadas com um "?". Nesses casos, os objetos da interface podem ou não definir essas propriedades.

```
1 interface Pessoa {
2   nome: string;
3   sobrenome?: string;
4   idade: number;
5 }
6
7 let dev1: Pessoa = {
8   nome: 'Iuri',
9   sobrenome: 'Silva',
10  idade: 22
11 } // OK
12
13 let dev2: Pessoa = {
14   nome: 'Eiji',
15   idade: 21
16 } // OK
```

No exemplo acima, a propriedade departamento está marcado com ?, então os objetos de Empregado podem ou não incluir esta propriedade.

Estendendo Interfaces

As interfaces podem estender uma ou mais interfaces, isso é, uma interface pode ter as propriedades declaradas em uma outra interface. Isso torna as interfaces e suas propriedades mais flexíveis e reutilizáveis.

```
1 interface Pessoa {
2   nome: string;
3   idade: number;
4 }
5
6 interface Pessoa2 extends Pessoa {
7   front: boolean;
8 }
9
10 let dev: Pessoa2 = {
11   nome: 'Iuri',
12   idade: 22,
13   front: true
14 }
```

No exemplo, a interface Pessoa2 estende a interface Pessoa. Portanto, os objetos de Pessoa2 devem incluir todas as propriedades e métodos da Pessoa interface.