

TypeScript

João Luiz Caieiro Borges Maravelli
Victor Ramos

Histórico

TypeScript é uma linguagem de programação extremamente nova, desenvolvida a partir de 2010 e sendo oficialmente lançada em 2012 pela Microsoft, sediada nos Estados Unidos, com o objetivo de melhorar a produtividade de quem trabalha com JavaScript. Seu desenvolvimento foi liderado principalmente pelo Dinamarquês Anders Hejlsberg, desenvolvedor de linguagens muito famosas e importantes como C# e Pascal.

Como o TypeScript é basicamente uma forma melhorada do JavaScript, sua popularização foi bem rápida, se tornando em 2018, a 4ª linguagem de programação mais amada pelo público em uma pesquisa feita pelo site Stack Overflow.

Figura 1 – Principal desenvolvedor do TypeScript, Anders Hejlsberg



Fonte: https://en.wikipedia.org/wiki/Anders_Hejlsberg

Motivação

Nos últimos anos, o desenvolvimento em JavaScript se tornou extremamente comum, por ser uma linguagem muito boa com diversas vantagens, com isso, foi notado um grande aumento no número de aplicações de grande escala sendo criadas em JavaScript, algo que era um problema, pois a criação de aplicativos grandes em JavaScript é uma tarefa difícil.

Este problema com o JavaScript foi o que surgiu de motivação para a criação do TypeScript, que tem como principal objetivo resolver o problema com relação a grandes projetos em JS, uma vez que esta tarefa era muito confusa e difícil.

Figura 2 – Logo das Linguagens JavaScript e TypeScript lado a lado



Fonte:

<https://leandro-oliveira.medium.com/javascript-vs-typescript-qual-a-diferen%C3%A7a-entre-os-dois-ed8c1cf83032>

Estrutura da linguagem

Variáveis

O TypeScript possui três formas principais de se declarar variáveis, sendo elas:

Declaração “Var”:

Uma variável declarada com ‘var’ se torna acessível para todo o programa, não sendo limitada ao que somos acostumados, por exemplo:

Figura 3 – Declaração de uma Variável ‘Var’:

```
1  var umaVariavel:number = 1;  
2
```

Fonte: Feita pelo autor.

Declaração “Let”:

Uma variável declarada com ‘let’ é uma variável que se torna acessível apenas para sua própria função onde foi declarada, resolvendo o problema da declaração ‘Var’.

Figura 4 - Declaração de uma Variável ‘Let’:

```
1  let umaVariavel:number= 1;  
2
```

Fonte: Feita pelo autor.

Declaração “Const”:

As variáveis declaradas com ‘const’ são variáveis muito parecidas com o ‘let’ em questão de seu funcionamento, porém, após um valor ser atribuído a mesma, isso não pode ser alterado. Também tem a característica de não poder ser declarada sem ser inicializada.

Figura 5 - Declaração de uma constante ‘Const’:

```
1  const PI:number = 3.1415;  
2
```

Fonte: Feita pelo autor.

Tipos de Dados

Number:

O tipo ‘number’ é associado a variáveis que têm como valor números, sejam eles inteiros ou flutuantes. Isto ocorre porque, para o TypeScript, todos os números são considerados flutuantes.

Figura 6 - Declaração de uma variável tipo “Number”:

```
1  var umNumero:number = 1;  
2
```

Fonte: Feita pelo autor.

Boolean:

O tipo 'boolean' é associado a variáveis que podem assumir os valores de verdadeiro e falso.

Figura 7 - Declaração de uma variável tipo "Boolean":

```
3  var umBool:boolean = true;  
4
```

Fonte: Feita pelo autor.

String:

O tipo String é associado a variáveis que assumem valores de texto.

Figura 8 - Declaração de uma variável tipo "String":

```
5  var umaString:string = "Uma String aqui!";  
6
```

Fonte: Feita pelo autor.

Array:

Array é um método para salvar diversas informações do mesmo tipo na mesma variável, sendo identificados pela sua posição no momento da declaração. Podem ser feitos arrays com qualquer tipo primitivo, seja ele 'Boolean', 'String' ou 'Number'.

Figura 9 - Declaração de dois arrays:

```
7  var umArray:number[] = [1, 2, 3, 4];  
8  var outroArray:boolean[] = [true, false, true, true];  
9
```

Fonte: Feita pelo autor.

Tuple:

Tuple é um tipo que permite com que seja possível fazer um array com um número fixo de elementos, em que os tipos são conhecidos.

Figura 10 - Declaração de uma variável em Tuple:

```
10 var pessoa: [string, number, string, boolean];  
11 pessoa = ["Victor", 18, "469.424.587-63", true];  
12
```

Fonte: Feita pelo autor.

Neste exemplo, o Tuple está sendo utilizado para guardar o nome, idade, CPF e um valor verdadeiro sobre uma pessoa.

Enum:

Enum é um tipo que permite que demos nome para variáveis números, por exemplo, atribuir um número aos dias da semana, daí vem seu nome.

Figura 11 - Declaração de uma variável do tipo Enum:

```
13 enum chamada{  
14     Adalberto,  
15     Bruna,  
16     João,  
17     Larissa,  
18     Vagner,  
19 }  
20
```

Fonte: Feita pelo autor.

Neste Exemplo, a saída para os valores seria: '1' para Adalberto, '2' para Bruna, '3' para João e assim por diante.

Unknown:

O Unknown é utilizado quando não se sabe qual é o tipo do valor que a variável receberá, desta forma, a variável aceitará qualquer tipo de valor.

Figura 12 - Declaração de uma variável tipo "Number":

```
21 var naoSei:unknown = true;
22 naoSei = "legal";
23 naoSei = 23;
24
```

Fonte: Feita pelo autor.

Na imagem acima, podemos ver que o compilador não aponta nenhum tipo de erro, mesmo após serem atribuídos valores de tipos diferentes a variável, pois o tipo ‘Unknown’ permite a atribuição de qualquer tipo durante o programa.

Any:

O tipo Any é muito semelhante ao Unknown, porém, sua diferença está em poder acessar propriedades que ainda não existem, e o compilador não irá verificar sua existência ou tipo.

Figura 13 - Declaração de uma variável tipo “Number”:

```
25 var qualquer:any = 70;
26 qualquer.podeSer = 12;
27
```

Fonte: Feita pelo autor.

Como pode-se ver pela imagem, mesmo não existindo um “qualquer.podeSer”, o compilador não aponta um erro, isto ocorre por causa da atribuição do tipo ‘Any’, o erro seria apontado caso o tipo declarado fosse ‘Unknown’.

Controle de Fluxo e Operadores

Controles de Fluxo são estruturas que alteram a ordem em que o programa é executado, Estruturas de Controle e Estruturas Condicionais são categorias de estruturas que têm este tipo de função dentro do programa, e são baseadas em condições construídas por operadores.

Operadores:

Suas funções são atribuir, comparar e fazer operações aritméticas para que seja possível a construção do programa, logo, são cruciais para se fazer qualquer programa.

Entre os tipos de Operadores, estão os **Operadores de Atribuição**, utilizados para atribuir dados para um operando.

Figura 14 - Operadores de Atribuição:

Nome	Operador	Operação a ser Feita
Atribuição	<code>x = y</code>	<code>x = y</code>
Atribuição de adição	<code>x += y</code>	<code>x = x + y</code>
Atribuição de subtração	<code>x -= y</code>	<code>x = x - y</code>
Atribuição de multiplicação	<code>x *= y</code>	<code>x = x * y</code>
Atribuição de divisão	<code>x /= y</code>	<code>x = x / y</code>
Atribuição de resto	<code>x %= y</code>	<code>x = x % y</code>
Atribuição exponencial	<code>x **= y</code>	<code>x = x ** y</code>

Fonte: Feito pelo Autor.

Existem também os **Operadores de Comparação**, operadores estes que tem como função, executar uma comparação, e então, retornar um valor booleano, ou seja, verdadeiro ou falso.

Figura 15 - Como utilizar um Operador de Comparação:

```

28  var comparar:number = 10;
29
30  comparar > 9 //Retorna verdadeiro, pois a comparação é verdadeira
31  comparar > 13 //Retorna falso, pois a comparação é falsa
32

```

Fonte: Feito pelo Autor

Estes são os Operadores de Comparação do TypeScript:

Figura 16 - Operadores de Comparação:

Nome	Operador	Retorna Verdadeiro Se:
Igual	<code>x == y</code>	Os operando forem iguais
Não Igual	<code>x != y</code>	Os operandos forem diferentes
Estritamente igual	<code>x === y</code>	Os operandos forem iguais e do mesmo tipo
Estritamente não igual	<code>x !== y</code>	Os operando não forem iguais ou não forem do mesmo tipo
Maior	<code>x > y</code>	O operando da esquerda for maior que o da direita
Maior ou igual	<code>x >= y</code>	O operando da esquerda for maior ou igual ao da direita
Menor	<code>x < y</code>	O operando da esquerda for menor que o da direita
Menor ou igual	<code>x <= y</code>	O operando da esquerda for menor ou igual ao da direita

Fonte: Feito pelo Autor.

Além destes dois tipos de operadores, existem também os **Operadores Aritméticos**, que tem como função alterar valores através de operações matemáticas.

Figura 17 - Operadores Aritméticos:

Nome	Operador	Exemplo de Resultado
Soma	(+)	3 + 2 = 5
Subtração	(-)	3 - 2 = 1
Multiplicação	(*)	3 * 2 = 6
Divisão	(/)	3 / 2 = 1,5
Módulo (Resto da Divisão)	(%)	12 % 5 = 2
Incremento	(++)	3++ = 4
Decremento	(--)	3-- = 2
Negação	(-)	Se x = 3, então -x = -3
Exponencial	(**)	3 ** 2 = 9

Fonte: Feito pelo Autor.

Por fim, os **Operadores Lógicos** são aqueles que retornam um valor booleano após avaliar uma condição.

Figura 18 - Operadores Lógicos:

Nome	Utilização de Operador	Exemplo de Resultado
AND	cond1 && cond2	Retorna verdadeiro caso as duas expressões sejam verdadeiras
OU	cond1 cond2	Retorna verdadeiro caso uma das duas expressões sejam verdadeiras
NOT	!cond1	Retorna o valor contrário da expressão

Fonte: Feito pelo Autor.

Existem também, o “**Conector de String**”, que age praticamente ‘Somando’ uma String com outra para formar uma única, por exemplo:

Figura 19 - Conector de String:

```
33  var somaString:string = "uma String" + "outra String";
34  // Resulta em "Uma String outra String"
35
```

Fonte: Feito pelo Autor.

Estruturas Condicionais:

Estruturas Condicionais são estruturas que fazem com que o programa execute comandos apenas se uma certa condição for cumprida, em TypeScript existem duas estruturas condicionais, sendo elas:

Estrutura Condicional: if

Nesta estrutura de repetição, o programa entrará na estrutura e ficará por lá enquanto a condição for comprida. Existem três variações desta estrutura condicional, sendo elas: if (unidimensional), if...else (bidirecional) e else...if (multidirecional).

Figura 20 - Estrutura if:

```
36  ✓ if (comparar == 10){  
37      //comandos aqui só serem executados se o valor de comparar for 10  
38  }  
39
```

Fonte: Feito pelo Autor.

Figura 21 - Estrutura if...else:

```
40  ✓ if (comparar == 10){  
41      //comandos aqui serem executados se a condição for verdadeira  
42  ✓ } else {  
43      //comandos aqui serem executados se a condição for falsa  
44  }  
45
```

Fonte: Feito pelo Autor.

Figura 22 - Estrutura else...if:

```
46  if (comparar == 10){  
47      //comandos aqui serem executados se comparar = 10  
48  } else if (comparar == 11){  
49      //comandos aqui serem executados se comparar = 11  
50  } else if (comparar == 12){  
51      //comandos aqui serem executados se comparar = 12  
52  } else {  
53      //comandos aqui serem executados se comparar não satisfazer nenhuma condição anterior  
54  }  
55
```

Fonte: Feito pelo Autor.

Estrutura Condicional: Switch

Na estrutura condicional Switch, o programa receberá uma expressão, e tentará associá-la a um dos ‘casos’ listados, executando o comando caso consiga fazer a associação.

Figura 23 - Estrutura Switch:

```

56  switch (comparar){
57      case 10:
58          //executar se comparar = 10
59          break;
60      case 11:
61          //executar se comparar = 11
62          break;
63      case 12:
64          //executar se comparar = 12
65          break;
66      default:
67          //executar se nenhum caso funcionar
68          break;
69  }

```

Fonte: Feito pelo Autor.

Estruturas de Repetição:

Estruturas de repetição são estruturas que fazem com que certa parte do código seja repetida um certo número de vezes, este número de vezes é definido com uma condição, e então, o código é repetido enquanto ele satisfazer esta condição. Existem duas estruturas de repetição em TypeScript, sendo elas: For e While.

Estrutura de Repetição: For

Na estrutura de repetição for, será dada uma condição e um escalonamento em relação a um contador, ou seja, o programador saberá quantas vezes esta seção será repetida durante a execução do programa.

Figura 24 - Estrutura For:

```

71  for (let contador:number = 0; contador<=10; contador++){
72      console.log(contador + " X " + "10" + " = " + (contador*10));
73      //Este comando será executado 10 vezes e imprimirá toda a tabuada do 10
74  }
75

```

Fonte: Feito pelo Autor.

Estrutura de Repetição: While

Diferente do ‘for’, a estrutura de repetição while é utilizada quando o programador não sabe quantas vezes a repetição irá acontecer dentro do programa, por exemplo, desta forma, o usuário do programa pode decidir quando certa repetição irá acontecer.

Ela funciona testando uma condição booleana a cada repetição, parando de repetir apenas quando tal condição se mostrar “falsa”. Esta estrutura de repetição pode ser dividida em duas, o while e o do...while.

Figura 25 - Estrutura While:

```
78 while (condicao < 20){  
79     console.log(condicao);  
80     //aqui deverá existir uma forma de tornar a condicao falsa, como por exemplo:  
81     condicao++;  
82     //neste caso, a repetição ocorrerá 20 vezes  
83 }  
84
```

Fonte: Feito pelo Autor.

A diferença entre while e do...while, é que neste segundo, a primeira execução do código ocorrerá indiferente da condição ser verdadeira ou não, a condição só determinará se existirá mais repetições do código.

Figura 26 - Estrutura Do...While:

```
78 do{  
79     console.log(condicao);  
80 } while (condicao < 20);  
81  
82 //Destá forma, mesmo a condição sendo falsa já na primeira execução  
83 //O código ainda executará a ação na primeira vez.  
84
```

Fonte: Feito pelo Autor.

Subprogramas

Subprograma é uma forma de construir uma abstração do programa, e então, utilizá-lo várias vezes sem ficar repetindo o código, tornando a programação muito mais simples e fácil, além de ser extremamente importante, principalmente em códigos grandes, o que, é justamente o foco do TypeScript, tornar a criação de códigos grandes mais fáceis.

Em TypeScript, os subprogramas recebem seus parâmetros por posição, e sua construção se dá por: **'function nomeFuncao(parametros:tipo):tipo{'** como no exemplo a seguir:

Figura 27 - Exemplo de Subprograma:

```
1  function subprograma(x:number, y:number):number{
2  |      return x+y;
3  |  }
4
5  console.log(subprograma(3,4));
6  //a saída será: 7
7
8  console.log(subprograma(7,2));
9  //a saída será: 9
10
```

Fonte: Feito pelo Autor.

Em TypeScript, existem diversos recursos para funções que não existem em JavaScript, como a declaração de funções anônimas, parâmetros padrão, parâmetros opcionais e parâmetros rest.

Função anônima:

Funções anônimas são basicamente, funções sem nome, desta forma, como em algum momento precisaremos “chamar”, teremos sempre que atribuir uma variável a esta função, da seguinte forma:

Figura 28 - Função anônima:

```
1  let anonima= function (x:number, y:number) {
2  |      return x+y;
3  |  }
4
5  console.log(anonima(10,5));
6  |
```

Fonte: Feito pelo Autor.

Função com parâmetro opcional:

Uma função com parâmetro opcional, assim como o nome diz, é uma função que pode ou não receber certos parâmetros, como por exemplo, em uma função onde é obrigatório receber o nome de uma pessoa, porém não é obrigatório receber a idade, o

parâmetro 'idade' seria colocado como opcional, para isso, basta colocar um "?" antes de declarar o tipo da variável. Lembrando que, caso o parâmetro opcional não seja fornecido, ele retornará "undefined"

Figura 29 - Função com parâmetro opcional:

```
1 function paramOpc(nome:string, idade?:number):string{
2     return nome + idade;
3 }
4
5 console.log(paramOpc("Victor"));
6 //A saída será Victorundefined
7
```

Fonte: Feito pelo Autor.

Além disso, é necessário que, após um parâmetro opcional, só sejam declarados outros parâmetros opcionais, ou seja, não é possível declarar um parâmetro opcional primeiro, e depois um parâmetro obrigatório.

Função com parâmetro padrão:

Uma função com parâmetro padrão é uma função na qual, caso algum parâmetro não seja fornecido, ele será preenchido automaticamente com algum valor pré definido pelo programador.

Basicamente, funciona como uma Função com parâmetro opcional, porém, em vez do parâmetro retorna "Undefined" caso não seja fornecido, ele retornará um valor pré estabelecido.

Figura 30 - Função com parâmetro padrão

```
1 ✓ function paramOpc(nome:string, idade:number = 12):string{
2     return nome + idade;
3 }
4
5 console.log(paramOpc("Victor"));
6 //A saída será Victor12
7
```

Fonte: Feito pelo Autor.

Função com parâmetro rest:

Este tipo de parâmetro é utilizado quando você não sabe quantos valores serão passados para a função. Quando isto ocorre, é possível criar um array como parâmetro, e então ele receberá quantos valores forem passados, podendo inclusive, não ser passado nenhum parâmetro. Para isto, é necessário apenas adicionar “...” antes da declaração do parâmetro, e declará-lo como array. Este deve ser o último parâmetro da função.

Figura 31 - Função com parâmetro Rest:

```
1  function paramRest(nome:string, ...idade:number[]){
2
3      let result:number = 0;
4
5      for( let i:number = 0; i<idade.length;i++){
6          result = result + idade[i];
7      }
8      return nome + " " + result;
9  }
10
11  console.log(paramRest("Victor", 12, 13, 15, 30));
12  //saída: Victor 70
13
```

Fonte: Feito pelo Autor.

Recurso específico

Tipos: Union e Intersection

Union:

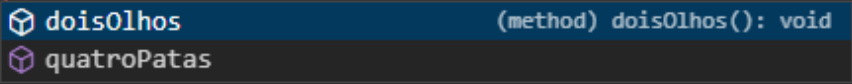
Union é um recurso específico da linguagem de TypeScript, que tem como função, falando de forma didática, ‘limitar’ o tipo Any. O tipo Union basicamente faz com que seja possível basicamente escolher um valor que seja um dentre vários tipos escolhidos. Para isso, utilizamos a barra vertical ‘|’ para separar os tipos escolhidos, por exemplo: number | boolean | string. Neste caso, o tipo pode ser tanto um Number, Boolean ou uma String. Algo importante a se ressaltar é que, ao utilizar um tipo Union, só conseguiremos acessar membros que são comuns a este tipo.

Figura 32 - Exemplo de Union:

```

1  interface cachorro{
2      quatroPatas(): void;
3      doisOlhos(): void;
4      latido(): void;
5  }
6
7  interface gato{
8      quatroPatas(): void;
9      doisOlhos(): void;
10     miado():void
11 }
12
13 declare function animal(): cachorro|gato;
14
15 let bixo = animal();
16 bixo.

```



Fonte: Feito pelo Autor.

Como podemos ver na imagem, o TypeScript deu como opção para completar, apenas os métodos “doisOlhos” e “quatroPatas”, isto ocorre pois os métodos “latido” e “miado” não fazem parte dos dois tipos do Union (“cachorro” e “gato”). Logo, este Union não consegue acessá-los.

Intersection:

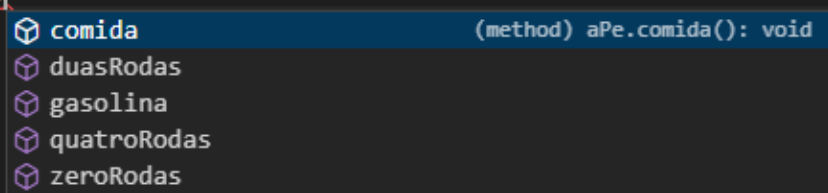
O tipo Intersection funciona de forma muito similar ao Union, porém, ao invés de conseguir acessar membros que fazem parte de todos os os tipos. Neste caso, poderemos acessar membros que fazem parte de pelo menos 1 dos tipos, basicamente combinando esses tipos. De forma simples de entender, o Intersection combina diversos tipos, tornando possível acessar todas as funções que ele necessitaria utilizando apenas um tipo. Para utilizar o Intersection, basta conectar os tipos desejados com “&”.

Figura 33 - Exemplo de Intersection:

```

1  interface carro{
2      quatroRodas(): void;
3      gasolina(): void;
4  }
5
6  interface moto{
7      duasRodas(): void;
8      gasolina(): void;
9  }
10
11 interface aPe{
12     zeroRodas(): void;
13     comida(): void;
14 }
15
16 declare function locomocao(): carro & moto & aPe;
17 let locomover = locomocao();
18
19 locomover.

```



Fonte: Feito pelo Autor.

Como podemos ver na imagem acima, o TypeScript permite que sejam relacionados os métodos “comida”, “duasRodas”, “gasolina”, “quatroRodas” e “zeroRodas”, mesmo eles sendo métodos de apenas um dos tipos listados pelo Intersection, isto ocorre pois sua função é justamente essa, apenas ‘somar’ todos os tipos listados.

Estes dois tipos listados são específicos do TypeScript, e são tipos mais avançados e de difícil compreensão para que não tem muita prática com a linguagem, porém, após um pouco de prática se torna muito intuitivo a utilização dos mesmos. Sendo estes, alguns dos recursos mais poderosos do TypeScript.

Avaliação das características da linguagem

A linguagem TypeScript tem diversas características muito positivas, não à toa, ela é, mesmo sendo muito nova, uma das linguagens mais amadas entre os programadores. Sua legibilidade e simplicidade são extremamente boas, ao ponto de

que, uma pessoa com conhecimento mediano em programação conseguiria facilmente pegar um programa em desenvolvimento e entendê-lo facilmente. Grande parte disso ocorre pois a linguagem possui uma boa ortogonalidade, o que faz com que a linguagem não tenha um excesso de regras, facilitando seu entendimento e utilização. Tendo também uma curva de aprendizado mediana. Caso a pessoa já tenha conhecimentos em JavaScript, é extremamente simples entender TypeScript.

Sua facilidade de escrita e confiabilidade também são muito bons, uma vez que a linguagem teve como objetivo em sua criação, criar os tipos e organizar-se para facilitar grandes projetos, a facilidade de escrever e ler TypeScript é muito grande, enquanto sua confiabilidade também é muito boa.

Instalação e configuração

Instalação:

A instalação do TypeScript é bem simples, sendo necessário apenas ter o Node.js em seu computador. Tendo o Node.js instalado em seu computador, basta instalar o TypeScript através do NPM, utilizando o seguinte comando no 'Prompt de Comando':

- 'npm install -g typescript' - Para Instalação Global
- 'npm install typescript --save-dev' - Para Instalação em um projeto específico.

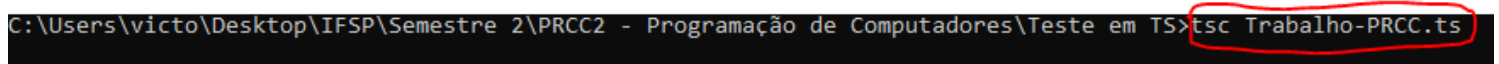
Também é possível instalar o TypeScript através de plugins no Visual Studio Code, para isso, basta instalar sua extensão que pode ser encontrada no 'Visual Studio Marketplace'.

Como Utilizar:

Para executar um programa em TypeScript, primeiro devemos transpilar o programa para JavaScript, para fazer isso, devemos digitar:

- 'tsc nomedoarquivo.ts', como do exemplo a seguir:

Figura 34 - Transpilação de código TypeScript:

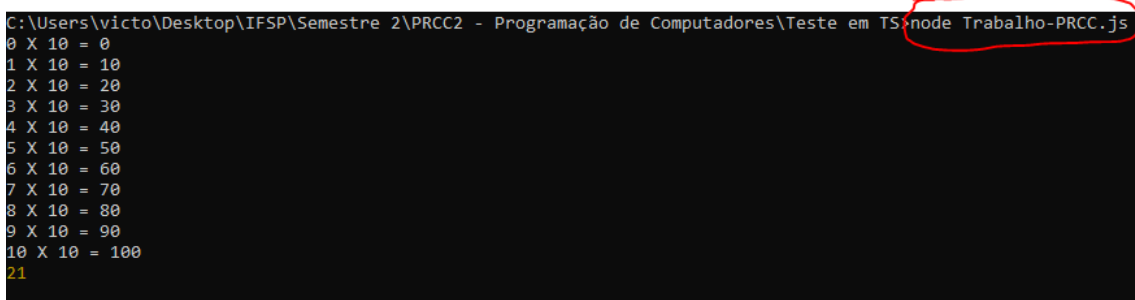


```
C:\Users\victo\Desktop\IFSP\Semestre 2\PRCC2 - Programação de Computadores\Teste em TS>tsc Trabalho-PRCC.ts
```

Fonte: Feito pelo Autor

Após isso, um arquivo com o mesmo nome, porém com a extensão .js será criado na mesma pasta. Agora, basta executá-lo da mesma forma que você utilizaria um arquivo JavaScript qualquer. como por exemplo, basta executá-lo com o Node no Prompt de Comando, da seguinte forma:

Figura 35 - Execução do arquivo JavaScript criado:



```
C:\Users\victo\Desktop\IFSP\Semestre 2\PRCC2 - Programação de Computadores\Teste em TS\node Trabalho-PRCC.js
0 X 10 = 0
1 X 10 = 10
2 X 10 = 20
3 X 10 = 30
4 X 10 = 40
5 X 10 = 50
6 X 10 = 60
7 X 10 = 70
8 X 10 = 80
9 X 10 = 90
10 X 10 = 100
21
```

Fonte: Feito pelo Autor.

Referências

MARQUES, Henrique. **O que é TypeScript e para que serve?** 2020. Disponível em: <https://marquesfernandes.com/tecnologia/o-que-e-typescript-e-para-que-serve/>. Acesso em: 24 Setembro 2022.

LUIS, Guilherme. **TypeScript, saiba tudo sobre a tecnologia.** 2019. Disponível em: <https://programathor.com.br/blog/typescript/>. Acesso em 24 Setembro 2022.

TypeScript. **TypeScript: JavaScript With Syntax for Types.** 2022. Disponível em: <https://www.typescriptlang.org/>. Acesso em 25 Setembro 2022.

NOLETO, Cairo. **TypeScript: o que é, principais conceitos e porquê usar!** 2020. Disponível em: <https://blog.betrybe.com/desenvolvimento-web/typescript/>. Acesso em 25 Setembro 2022

StackOverflow. **Stack Overflow Developer Survey 2020 .** 2020. Disponível em: <https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents>. Acesso em: 25 Setembro 2022.

Edson. **TypeScript Tutorial: Introdução Completa ao TypeScript.** 2016. Disponível em: <https://marquesfernandes.com/tecnologia/o-que-e-TypeScript-e-para-que-serve/>. Acesso em: 01 Outubro 2022.

FERNANDES. Diego. **TypeScript: Vantagens, mitos, dicas e conceitos fundamentais.** 2019. Disponível em: <https://blog.rocketseat.com.br/typescript-vantagens-mitos-conceitos/>. Acesso em: 01 Outubro 2022.

MethratOn. **Union Types and Intersection Types**. 2019. Disponível em:
<https://medium.com/@Methrat0n/union-types-and-intersection-types-50c41c9b61d6>. Acesso
em: 08 Outubro 2022

PIECHOCKI, Miłosz. **The meaning of union and intersection types**. 2019. Disponível em:
<https://codewithstyle.info/The-meaning-of-union-and-intersection-types/>. Acesso em: 09
Outubro 2022