



# Desenvolvimento WEB II

# TypeScript

Guia rápido

# Introdução ao TypeScript

- O TypeScript é um pré-processador (superset) de códigos JavaScript open source desenvolvido e mantido pela Microsoft. Ele foi desenvolvido pelo time do Anders Hejlsberg, arquiteto líder do time TypeScript e desenvolvedor do C#, Delphi e do Turbo Pascal. A sua primeira versão, a 0.8, foi lançada em 1 de outubro de 2012.
- Ele foi projetado para auxiliar no desenvolvimento de códigos simples até os mais complexos, utilizando os princípios de Orientação a Objetos, como classes, tipagens, interfaces, Generics etc.

# Introdução ao TypeScript

- Aqui chegamos à primeira dúvida de todo desenvolvedor que está iniciando com TypeScript: por que tipar o JavaScript?
- Essa, na realidade, é uma das vantagens de se trabalhar com esse pré-processador. A utilização de tipagem ajuda no momento de depuração (debug) do nosso código, prevenindo alguns possíveis bugs ainda em tempo de desenvolvimento.

# Introdução ao TypeScript

- Mas como ele funciona?
- Como os navegadores não interpretam código TypeScript, nós precisamos transpilar o nosso código para uma das versões ECMAScript.
- A figura a seguir demonstra o seu processo de compilação (transpiler). O arquivo .ts é o código desenvolvido em TypeScript, a engrenagem é o compilador, e o arquivo .js é o código final JavaScript.



# Instalação

- Para instalar o TypeScript, é necessário ter o Node.js e o seu gerenciador de pacotes, o NPM, instalados no seu computador. Caso você ainda não tenha, basta acessar o link <https://nodejs.org/en/> e baixar um executável de acordo com o seu sistema operacional. Para verificar se ele foi instalado corretamente ou qual é a versão que você tem instalada, basta abrir um terminal no seu computador e digitar o comando: `node -v` || `npm -v` .

```
PS C:\Users\saoavnia> node -v
v20.11.1
PS C:\Users\saoavnia> npm -v
10.2.4
PS C:\Users\saoavnia> |
```

# Instalação

- Ao utilizar o npm seguido de install , estamos passando para o gerenciador que ele deve instalar o pacote informado depois da instrução install .
- Um ponto importante para destacar neste momento é que, quando trabalhamos com pacotes npm , é comum termos alguns deles instalados em modo global, o que nos permite executar esse pacote de qualquer lugar do nosso computador. Para ter um pacote npm instalado em modo global, basta adicionar o -g antes do nome do pacote. Vamos instalar o TypeScript em modo global. Para isso, digite o comando: `npm install -g typescript` no seu terminal.

# Instalação

- Uma vez instalado, o compilador do TypeScript estará disponível executando o comando `tsc`. Para verificar a versão instalada, basta digitar o comando `tsc -v` dentro de um terminal.

```
PS C:\Users\saoavnia> npm install -g typescript  
  
added 1 package in 2s  
PS C:\Users\saoavnia> tsc -v  
Version 5.5.4  
PS C:\Users\saoavnia> |
```



# Executando manualmente o TypeScript

- Com o TypeScript instalado em modo global, crie um novo diretório no seu computador para organizar os exemplos desta seção. Dentro desse diretório, crie um arquivo chamado: `meuprimeiro.ts` .
- Para os próximos passos será necessária a utilização de um editor de textos. Vou utilizar o Visual Studio Code pela sua integração com o TypeScript, mas todos os exemplos aqui demonstrados funcionarão em qualquer editor de textos.

# Executando manualmente o TypeScript

- Com o seu editor de textos aberto, abra o arquivo `meuprimeiro.ts` nele e em seguida atualize-o com o seguinte trecho de código:

```
const a : string = 'Hello World';  
console.log(a);
```

- Abra um terminal no seu computador, navegue até o diretório em que você criou o arquivo `meuprimeiro.ts` e execute o comando `tsc meuprimeiro.ts`

# Executando manualmente o TypeScript

- Observe que será criado um novo arquivo chamado `meuprimeiro.js` na raiz de sua pasta.
- Agora, para validar o passo anterior, execute o comando

*`node meuprimeiro.js` no seu terminal.*

- Caso tudo esteja OK, você deve receber a seguinte mensagem no seu terminal: Hello World.

# Entendendo o compilador do TypeScript

- O compilador do TypeScript é altamente configurável. Ele nos permite definir o local onde estão os arquivos .ts dentro do nosso projeto, o diretório de destino dos arquivos transpilados, a versão ECMAScript que será utilizada, o nível de restrição do verificador de tipos e até se o compilador deve permitir arquivos JavaScript.
- Cada uma das opções de configuração pode ser passada para um arquivo chamado tsconfig.json . Esse é o principal arquivo de configuração do TypeScript

# Entendendo o compilador do TypeScript

- Para criar esse arquivo dentro de um novo projeto, basta executar o comando `tsc --init` . Isso vai criar um arquivo na raiz do seu projeto com algumas configurações padrões, como `target` , `module` entre outras. A seguir você tem uma imagem demonstrando esse arquivo com as configurações default dele.

```
1 {
2   "compilerOptions": {
3     /* Visit https://aka.ms/tsconfig to read more about this file */
4
5     /* Projects */
6     // "incremental": true,           /* Save .tsbuildinfo to speed up compilation */
7     // "composite": true,            /* Enable constraints that allow a TypeScript compiler to be used by multiple compilers.
8     //                                /* Specify the path to the TypeScript compiler's .tsbuildinfo file */
9     // "tsBuildInfoFile": "./.tsbuildinfo",
10    // "disableSourceOfProjectReferenceRedirect": true, /* Disable preferring source files instead of
11    // "disableSolutionSearching": true,           /* Opt a project out of the building process when
12    // "disableReferencedProjectLoad": true,        /* Reduce the number of projects loaded by this
13
14    /* Language and Environment */
15    "target": "es2016",                /* Set the JavaScript target version. Set this to the highest
16    // "lib": [],                        /* Specify a set of built-in lib files to be included in your
```

# Entendendo o compilador do TypeScript

- Para que você possa entender melhor como funciona o compilador, vamos criar um novo arquivo TypeScript.
- No mesmo diretório que você criou o seu arquivo de configurações no passo anterior, crie um arquivo com a extensão `.ts` . Para esse exemplo, eu criarei um arquivo chamado `index.ts` , mas você pode escolher um nome de sua preferência.
- Um ponto importante que devemos sempre lembrar no momento de criação de novos arquivos, é respeitar a convenção que os desenvolvedores JavaScript já utilizam hoje, a utilização do camelCase . Exemplo: `myControl.ts` .

# Entendendo o compilador do TypeScript

- Com o seu arquivo TypeScript criado, atualize-o com o seguinte trecho de código:

```
var languages : Array = [];  
languages.push("TypeScript");  
languages.push(3)
```

# Entendendo o compilador do TypeScript

- Com o seu arquivo TypeScript criado, atualize-o com o seguinte trecho de código:

```
var languages : Array = [];  
languages.push("TypeScript");  
languages.push(3)
```

- Analisando esse trecho de código, nós criamos um array de string e estamos adicionando a ele uma string com o valor TypeScript e um número com o valor 3. Será que o compilador vai transpilar esse código? Note que nós temos um array de string e estamos tentando passar um número para ele.



# Entendendo o compilador do TypeScript

- Para validar se o compilador vai transpilar esse código, abra um terminal no seu computador, navegue até o diretório em que você criou o arquivo de configurações e o seu arquivo .ts e, em seguida, execute o comando `tsc nomeDoSeuArquivo.ts` , trocando "nomeDoSeuArquivo" pelo nome que você deu, no meu caso foi "index". Note que o compilador transpilou o código e gerou o arquivo index.js na mesma pasta dos outros arquivos, mesmo entendendo que tem um erro no código. Agora vem aquela dúvida: o TypeScript não foi desenvolvido para prevenir erros como esse em tempo de desenvolvimento?

# Entendendo o compilador do TypeScript

- A resposta é sim. O TypeScript ajuda a prevenir erros como esse em tempo de desenvolvimento, mas para que ele possa alertar ou barrar a geração de arquivos com erro, como o do exemplo anterior, nós devemos informar ao compilador como nós queremos trabalhar com ele através do arquivo `tsconfig.json`. Para isso, abra o seu arquivo e adicione a seguinte configuração nele:

```
67 // "newLine": "crlf", /* Set the newline character for emitting files */
68 // "stripInternal": true, /* Disable emitting declarations that have "@internal" tag */
69 // "noEmitHelpers": true, /* Disable generating custom helper functions like __extends */
70 "noEmitOnError": true, /* Disable emitting files if any type checking errors occur */
71 // "preserveConstEnums": true, /* Disable erasing 'const enum' declarations */
72 // "declarationDir": "./", /* Specify the output directory for generated declaration files */
73
```

- Essa configuração fará com que o compilador analise o seu código e, caso ele encontre algum erro, não crie o arquivo `.js`.

# Automatizando o compilador

- O nosso processo está muito manual. Hoje o nosso exemplo tem somente um arquivo, mas quando estivermos trabalhando em um projeto com mais arquivos .ts será necessário executar o comando `tsc` + nome do arquivo em cada um deles? A resposta é não.
- O TypeScript tem um parâmetro que nós podemos passar para o compilador ficar verificando todos os arquivos do projeto em tempo real, dando feedbacks e, em caso de sucesso, transpilando todos de uma vez. Para utilizar esse parâmetro é bem simples, basta adicionar o `-w` na frente da instrução `tsc`.

# Automatizando o compilador

- Voltando para o nosso projeto, exclua os arquivos com a extensão .js, e ainda com o seu terminal aberto, execute o comando `tsc -w`.
- Observe que agora o compilador emitirá uma mensagem de erro e nenhum dos arquivos foi transpilado para javascript

```
index.ts:3:16 - error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.
```

```
3 languages.push(3)
```

```
[16:08:40] Found 1 error. Watching for file changes.
```

# Automatizando o compilador

- Remova a linha com erro do arquivo criado, no meu exemplo index.ts e clique em salvar (pressione ctrl+s)
- Podemos observar que como o arquivo não apresenta mais erros, ambos foram transpilados automaticamente para Javascript

# Entrada de dados

- Vamos usar o prompt-sync para receber os dados do usuário pelo terminal.  
<https://www.npmjs.com/package/prompt-sync>
- Precisamos instalar o pacote do prompt-sync digitando no terminal o seguinte comando:  
`npm i prompt-sync`
- Para usar o prompt-sync, vamos importar o pacote e criar uma instância do objeto:  

```
import promptSync = require('prompt-sync');  
var prompt = promptSync();  
var n = prompt('Mensagem');
```

# Conhecendo os Types

- O TypeScript é uma linguagem fortemente tipada, ou seja, devemos definir o tipo de dados da variável quando a declaramos.
- Podemos declarar variáveis utilizando as palavras chaves: `const`, `let`, `var`
- Um escopo no JavaScript é dado por funções e não por blocos, e a palavra reservada `var` permite que a variável declarada dentro de um escopo seja acessada de qualquer ponto de dentro do código.

# Conhecendo os Types

- A palavra reservada **let** é usada para declarar variáveis com escopo de bloco. Seu comportamento é idêntico ao var quando declarada fora de uma function , isto é, ela fica acessível no escopo global. Mas, quando declarada dentro de qualquer bloco, seja ele uma function , um if ou um loop , ela fica acessível apenas dentro do bloco (e sub-blocos) em que foi declarada.
- A palavra reservada **const** é usada para declarar variáveis readonly, isto é, a variável não pode ter o seu valor alterado, seu estado é imutável. Assim como as variáveis declaradas como **let** e **const** também ficam limitadas a um escopo.



# Conhecendo os Types

- A sintaxe para declaração de variáveis em TypeScript é:

**nomedavariavel** : **tipo**

Agora entrando nos types, veremos os tipos de dados suportados pela linguagem:

- **Boolean**: suporta dois tipos de valores: **true** ou **false**
- **Number**: No TypeScript, todos os valores numéricos, como floating , decimal , hex , octal devem ser tipados como number .
- **Strings**: As strings armazenam valores do tipo texto. Diferente de outras linguagens de programação, no JavaScript/TypeScript nós podemos declarar uma string em aspas simples e aspas duplas

# Vamos praticar?

1) Crie um programa que leia um número na tela, e exiba seus 2 sucessores e 2 antecessores.

- Exemplo: numero = 5, antecessores 4,3, sucessores 6,7

# Conhecendo os Types

- **Arrays:** Como em outras linguagens de programação, nós declaramos um array no TypeScript utilizando as chaves [].

```
let numeros: number[] = [1, 2, 3];  
let textos : string[] = ["exemplo 1", "exemplo 2", "exemplo 3"];
```

- Ou nós podemos utilizar a palavra reservada Array<> ,

```
let numeros: Array<number> = [1, 2, 3];  
let textos: Array<string> = ["exemplo 1", "exemplo 2", "exemplo 3"];
```

- Para adicionar um novo item ao array, nós podemos utilizar a palavra reservada push .

# Conhecendo os Types

- Para adicionar um novo item ao array, nós podemos utilizar a palavra reservada push .

```
let numeros: Array<number> = [1, 2, 3];
let textos: Array<string> = ["exemplo 1", "exemplo 2", "exemplo 3"];

numeros.push(4);
textos.push("exemplo 3");

console.log(numeros);
console.log(textos);

//Resultado
[ 1, 2, 3, 4 ]
[ 'exemplo 1', 'exemplo 2', 'exemplo 3', 'exemplo 3' ]
```

# Conhecendo os Types

- **ReadonlyArray**: é um array que nos permite somente leitura. Ele remove todos os métodos de alteração de um array, como push , pop etc.

```
let numerosDaMega: ReadonlyArray<number> = [8, 5, 5, 11, 4, 28];  
numerosDaMega[0] = 12; // error!  
numerosDaMega.push(23); // error!  
numerosDaMega.pop(); // error!  
numerosDaMega.length = 100; // error!
```

# Vamos praticar?

2) Crie um programa que declare um array com os números de 1 até 10. A seguir, solicite que o usuário digite um número maior que 10 e adicione o número digitado no array. Mostre todos os dados do array na tela

# Conhecendo os Types

- **Tuplas:** As tuplas ou tuple no inglês, representam uma estrutura de dados simples semelhante a um array. A grande diferença entre eles é que nos arrays nós trabalhamos somente com um tipo de dado, enquanto com as tuplas temos diferentes tipos.

```
let list: [string, number, string] = ['string', 1, 'string 2'];
```

- Uma das novidades da versão 4.0 do TypeScript é a possibilidade de incluir nomes nos parâmetros das nossas tuplas. Essa é uma das novidades que, na minha opinião, pode ajudar a deixar o nosso código mais legível.

```
let list: [nome: string, idade: number, email: string] = ['Bill Gates',  
65, 'bill@teste.com'];
```

# Conhecendo os Types

- **Enum:** O enum nos permite declarar um conjunto de valores/constantes predefinidos. Existem três formas de se trabalhar com ele no TypeScript: Number. String. Heterogeneous.
- Numérico Os enums numéricos armazenam strings com valores numéricos. Nós podemos declará-los com um valor inicial

```
export enum DiaDaSemana {  
    Segunda = 1,  
    Terca = 2,  
    Quarta = 3,  
    Quinta = 4,  
    Sexta = 5,  
    Sabado = 6,  
    Domingo = 7  
}
```



# Conhecendo os Types

- Caso você não passe um valor inicial na declaração do seu enum , o compilador do TypeScript fará um autoincremento de +1 iniciando em 0 até o último elemento do enum .

```
export enum DiaDaSemana {  
    Segunda,  
    Terca,  
    Quarta,  
    Quinta,  
    Sexta,  
    Sabado,  
    Domingo  
}
```

# Conhecendo os Types

- Ou, no caso de passarmos um valor numérico inicial, ele fará como no passo anterior, incrementando +1 até o último elemento:

```
export enum DiaDaSemana {  
    Segunda = 18,  
    Terca,  
    Quarta,  
    Quinta,  
    Sexta,  
    Sabado,  
    Domingo  
}
```

# Conhecendo os Types

- Para acessar um valor dentro de um enum , nós podemos utilizar uma das formas a seguir:

```
let dia = DiaDaSemana[19]; // Terca
let diaNumero = DiaDaSemana[dia]; // 19
let diaString= DiaDaSemana["Segunda"]; // 18
```

# Conhecendo os Types

- Diferente dos enums numéricos, os enums do tipo string precisam iniciar com um valor.

```
export enum DiaDaSemana {  
    Segunda = "Segunda-feira",  
    Terca = "Terça-feira",  
    Quarta = "Quarta-feira",  
    Quinta = "Quinta-feira",  
    Sexta = "Sexta-feira",  
    Sabado = "Sábado",  
    Domingo = "Domingo",  
}
```

# Conhecendo os Types

- Para acessar os valores de um enum do tipo string, basta utilizar uma das formas a seguir:

```
console.log(DiaDaSemana.Sexta); //Sexta-feira  
console.log(DiaDaSemana['Sabado']); //Sábado
```

# Conhecendo os Types

- Os enums Heterogeneous são pouco conhecidos. Eles aceitam os dois tipos de valores: strings e números

```
export enum Heterogeneous {  
    Segunda = 'Segunda-feira',  
    Terca = 1,  
    Quarta,  
    Quinta,  
    Sexta,  
    Sabado,  
    Domingo = 18,  
}
```

# Conhecendo os Types

- E como acessar esses valores?

```
console.log(Heterogeneous.Segunda);  
console.log(Heterogeneous['Segunda']);
```

```
console.log(Heterogeneous['Terca']);  
console.log(Heterogeneous[1]);
```

```
console.log(Heterogeneous['Quarta']);  
console.log(Heterogeneous['Quinta']);  
console.log(Heterogeneous['Sexta']);  
console.log(Heterogeneous['Sabado']);  
console.log(Heterogeneous['Domingo']);
```

Resultado:

```
//Segunda-feira  
//Segunda-feira  
//1  
//Terca  
//2  
//3  
//4  
//5  
//18
```

# Conhecendo os Types

- E como acessar esses valores?

```
console.log(Heterogeneous.Segunda);  
console.log(Heterogeneous['Segunda']);
```

```
console.log(Heterogeneous['Terca']);  
console.log(Heterogeneous[1]);
```

```
console.log(Heterogeneous['Quarta']);  
console.log(Heterogeneous['Quinta']);  
console.log(Heterogeneous['Sexta']);  
console.log(Heterogeneous['Sabado']);  
console.log(Heterogeneous['Domingo']);
```

Resultado:

```
//Segunda-feira  
//Segunda-feira  
//1  
//Terca  
//2  
//3  
//4  
//5  
//18
```



# Conhecendo os Types

- **Union:** O union nos permite combinar um ou mais tipos. Sua sintaxe é um pouco diferente dos outros types, ele utiliza uma barra vertical para passar os tipos que ele deve aceitar.

Sintaxe: (tipo1 | tipo2 ...)

Para ficar mais claro, vamos a alguns exemplos:

```
let exemploVariavel: (string | number);  
exemploVariavel = 123;  
console.log(exemploVariavel);  
exemploVariavel = "ABC";  
console.log(exemploVariavel);
```

Resultado:

```
//123  
//ABC
```

# Conhecendo os Types

- **Any:** O any é um dos types que mais causa confusão quando nós estamos iniciando com o TypeScript. Como o TS é tipado e nós temos o Union em casos de mais de um valor, a maioria dos desenvolvedores iniciantes em TypeScript se pergunta: Por que utilizar o any? Para ficar mais claro como nós podemos utilizar o type any , vamos a um exemplo prático

```
let variavelAny: any = "Variável";  
variavelAny = 34;  
variavelAny = true;
```

# Tipando Funções

- O TypeScript também nos permite tipar as nossas funções informando para o compilador qual é o tipo que elas devem retornar. A seguir você tem um trecho de código com uma função que recebe dois parâmetros number e retorna uma string:

```
function calc(x: number, y: number): string {  
    return `resultado: ${x + y}`;  
}
```

# Tipando Funções

- O TypeScript também nos permite tipar as nossas funções informando para o compilador qual é o tipo que elas devem retornar. A seguir você tem um trecho de código com uma função que recebe dois parâmetros number e retorna uma string:

```
function calc(x: number, y: number): string {  
    return `resultado: ${x + y}`;  
}
```

# Tipando Funções

- No TypeScript, podemos tipar uma variável com o valor de uma função. Pegando o nosso exemplo anterior, vamos atribuir o valor da function calc a uma variável

```
function calc(x: number, y: number): string {  
    return `resultado: ${x + y}`;  
}
```

```
let resultado : string;  
resultado = calc(10,15);  
console.log(resultado);
```

```
Resultado:  
//resultado: 25
```

- Caso você tente atribuir esse valor a uma variável de um tipo diferente de string, o compilador retornará a mensagem de erro

# Tipando Funções

- No TypeScript, podemos tipar uma variável com o valor de uma função. Pegando o nosso exemplo anterior, vamos atribuir o valor da function calc a uma variável

```
function calc(x: number, y: number): string {  
    return `resultado: ${x + y}`;  
}
```

```
let resultado : string;  
resultado = calc(10,15);  
console.log(resultado);
```

```
Resultado:  
//resultado: 25
```

- Caso você tente atribuir esse valor a uma variável de um tipo diferente de string, o compilador retornará a mensagem de erro

# Tipando Funções

- O type **void** é bastante utilizado em funções. Diferente do type any , que espera o retorno de qualquer valor, o void passa para o compilador que aquela função não terá nenhum retorno.

```
function log(): void {  
    console.log('Sem retorno');  
}
```

# Tipando Funções

- Embora o `type never` tenha sido adicionado à versão 2.0 do TypeScript, ele é pouco conhecido pelos desenvolvedores. Esse `type` indica que algo nunca deve ocorrer. Para que você possa ter uma ideia de como utilizá-lo no seu dia a dia, vamos a alguns exemplos práticos. Nós podemos utilizá-lo em funções com exception

```
function verificandoTipo(x: string | number): boolean {  
    if (typeof x === "string") {  
        return true;  
    } else if (typeof x === "number") {  
        return false;  
    }  
  
    return fail("Esse método não aceita esse tipo de type!");  
}  
  
function fail(message: string): never { throw new Error(message); }
```



# Tipando Funções

- Nesse exemplo, nós estamos recebendo via parâmetro um type union (string e número), que deve retornar um boolean. Caso seja passado um valor que não seja uma string ou um número, a chamada entra em uma outra função chamada fail , que retornará uma exceção.
- Diferenças entre os tipos void e never:
  - O type void pode receber o valor null , que indica ausência de um objeto, ou undefined , que indica a ausência de qualquer valor. O type never não pode receber valor.

# Vamos praticar?

3) Crie uma função que receba dois valores numéricos e mostre na tela o resultado das seguintes operações:

- Soma
- Subtração
- Divisão
- Multiplicação

4) Crie uma função que receba dois valores numéricos, realize a soma desses valores e retorne o valor calculado para uma variável chamada total. A seguir imprima o valor dessa variável

# Estruturas de controle e repetição

- **if-else:**

```
let condition = true;

if (condition)
{
    console.log("a variável está com um valor true");
}
else
{
    console.log("a variável está com um valor false");
}
```

# Estruturas de controle e repetição

- **if-else-if:**

```
let perfil = "admin";

if (perfil == "superuser") {
    console.log("Super usuário");
}
else if (perfil == "admin") {
    console.log("Administrador");
} else {
    console.log("Usuário comum");
}
```

# Estruturas de controle e repetição

- **Operador ternário:** O operador condicional ternário avalia uma expressão booleana e retorna o resultado de uma das duas expressões, conforme ela é avaliada como true ou false. Exemplo validando 2 perfis:

```
let perfil = "admin";  
console.log(perfil == "superuser" ? "Super usuário" : "Administrador");
```

- Validando três perfis

```
let perfil = "admin";  
console.log(perfil == "superuser" ? "Super usuário" : perfil == "admin" ?  
"Administrador" : "Usuário comum");
```

# Estruturas de controle e repetição

- **switch:**

```
let perfil = "admin";

switch (perfil) {
  case "superuser":
    console.log("Super usuário");
    break;
  case "manager":
    console.log("Gerente");
    break;
  case "admin":
    console.log("Administrador");
    break;
  case "user":
    console.log("Usuário comum");
    break;
  default:
    console.log("sem perfil");
    break;
}
```

# Estruturas de controle e repetição

- **while:**

```
let condicao = true;

while (condicao)
{
    console.log('Carregando...')
}
```

# Estruturas de controle e repetição

- **do-while:**

```
let condicao = false;  
{  
  console.log('Carregando...')  
}  
while (condicao);
```



# Estruturas de controle e repetição

- **for**

```
var languages = ["C#", "Java", "JavaScript", "TypeScript"];

for (let i = 0; i < languages.length; i++) {
  console.log(languages[i]);
}
```

# Estruturas de controle e repetição

- **foreach**

```
var languages = ["C#", "Java", "JavaScript", "TypeScript"];

languages.forEach(element => {
    console.log(element);
});
```

# Vamos praticar?

- 5) Crie um programa que leia dois valores (x e y) representando um intervalo. Em seguida, leia um novo valor (z) e verifique se z pertence ao intervalo [x, y].
- 6) Crie um programa que leia um número do usuário e informe se o número é par ou ímpar.
- 7) Escreva um programa que leia o ano de nascimento de uma pessoa e apresente uma linha do tempo, mostrando cada ano de vida da pessoa e quantos anos ela tinha em cada ano até o ano atual.

# Vamos praticar?

- 8) Escrever um programa que lê um vetor N(20) e o escreve. Troque, a seguir, o 1º elemento com o último, o 2º com o penúltimo etc. até o 10º com o 11º e escreva o vetor N assim modificado.
- 9) Crie uma função de conversão entre as temperaturas Celsius e Fahrenheit. Primeiro o usuário deve escolher se vai entrar com a temperatura em Célsius ou Fahrenheit, depois a conversão escolhida é realizada através de um comando SWITCH. Se C é a temperatura em Célsius e F em Fahrenheit, as fórmulas de conversão são:  $C = 5 \cdot (F - 32) / 9$   $F = (9 \cdot C / 5) + 32$
- 10) Escreva um programa que leia as notas dos alunos de uma disciplina e armazene em um array. (A quantidade de alunos deve ser informada pelo usuário) e informe quantos alunos estão abaixo da média e quantos estão na média. (Considere a nota sendo um inteiro de 0 a 100 e a média 60)