COM222 Desenvolvimento de Sistemas na Web

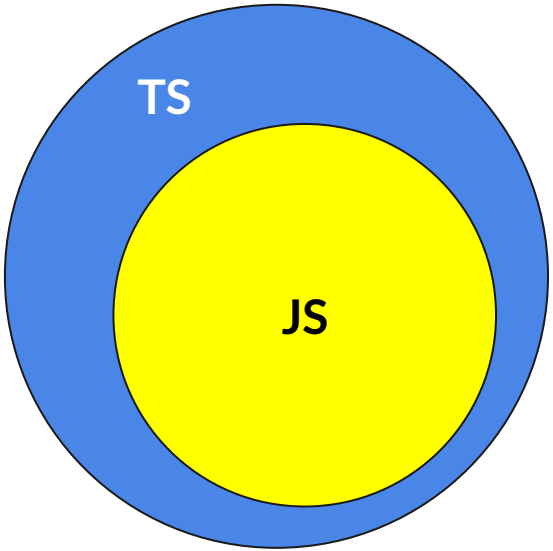
Aula13: Typescript

O que é Typescript?

- TypeScript é um **superconjunto** de JavaScript desenvolvido pela Microsoft que **adiciona recursos a linguagem**
- É uma ferramenta de **desenvolvimento!**
    - Código Typescript é transpilado para Javascript e, assim, pode rodar no **Node JS** ou nos **navegadores web**

Conceito de super conjunto

Todo código **Javascript** é um código **Typescript** válido, mas *nem todo código* **Typescript** é um código **Javascript** válido



Exemplo

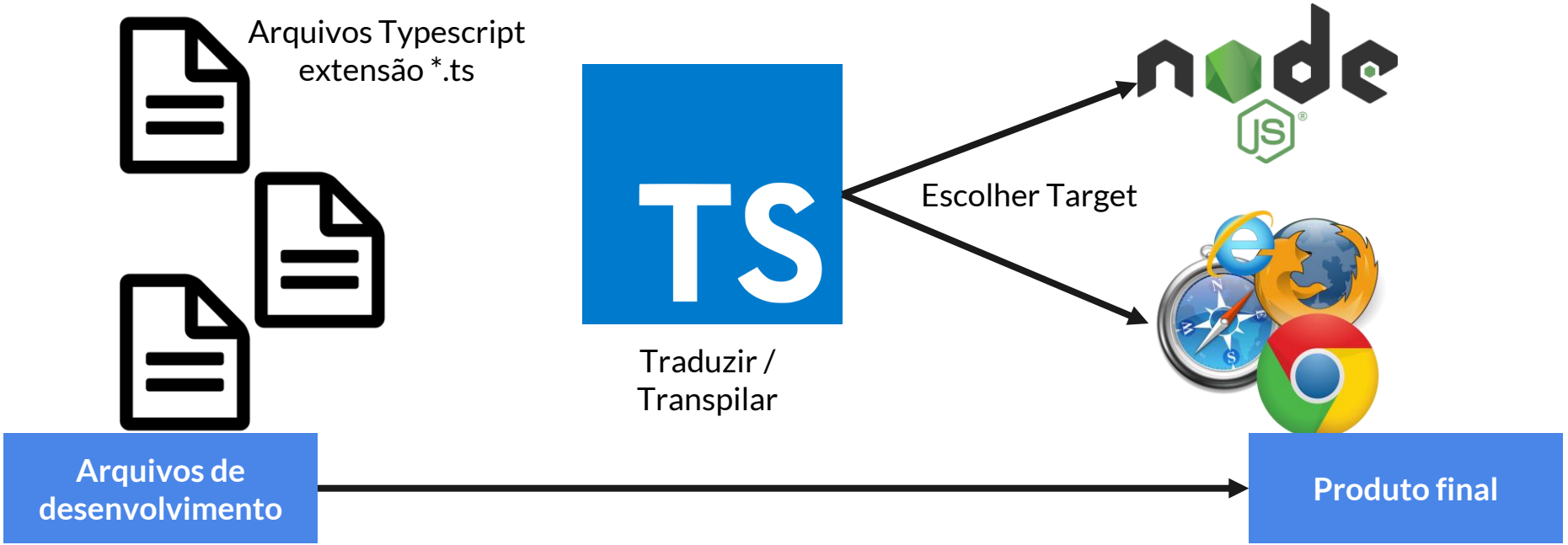
- Válido em ambos

- Válido somente no Typescript

```
function FalaOi(nome) {
    return "Hello, " + nome;
}

// somente typescript
function FalaOi(nome: string): string {
    return "Hello, " + nome;
}
```

O que é Typescript?



Por quê Typescript

- O Javascript é perigosamente flexível

Quais são as vantagens do Typescript?

- Muito parecido com o C#
- Facilita manutenção de projetos grandes
- É relativamente fácil migrar do Javascript para o Typescript
- Leva conceitos de POO para o Javascript e garante compatibilidade
  - es2015
- Facilita tipagem e intellisense
  - Igual o netbeans faz com o Java
  - As IDEs trabalham muito bem com typescript, principalmente o VS Code

O caso do angular

- O Angular 1 (ou AngularJS) foi feito em **Javascript**
- O **Angular2+** foi o AngularJS completamente reescrito com **Typescript** justamente para aproveitar a robustez e os novos recursos
- É possível (apesar de difícil e não recomendado) converter uma aplicação construída com AngularJS para algo parecido com o Angular2+

Instalando o Typescript

npm install -g typescript

O que precisa ter?

- NodeJS - https://nodejs.org/en/
- Abrir o terminal e garantir que os comandos **node** e **npm** estão funcionando
- Typescript instalado
  - npm install -g typescript
- **[Opcional]** Visual Studio Code
- **[Opcional]** http-server
  - npm install -g http-server

Brincando no browser

Somente para fins didáticos

- Dificilmente isso vai acontecer na prática
- Criar os arquivos **index.ts** e **index.html**
- Vamos para o código

index.ts

```
function FalaOi(nome) {
    return "Hello, " + nome;
}

let nome = 'Fulano'

document.body.textContent = FalaOi(nome);
```

No terminal

```
tsc index.ts --target es2015
```

```
http-server .
```

Para quem instalou o http-server

apareceu um index.js

```
function FalaOi(nome) {
    return "Hello, " + nome;
}
let nome = 'Fulano';
document.body.textContent = FalaOi(nome);
```

É possível mudar o target

```
tsc index.ts --target es6
```

index.html

```
<!DOCTYPE html>
<html>
    <head><title>TypeScript Greeter</title></head>
    <body>
        <script src="index.js"></script>
    </body>
</html>
```

Tudo de novo, agora no NodeJS

index.ts

```
function FalaOi(nome) {
    return "Hello, " + nome;
}

let nome = 'Fulano'

console.log(FalaOi(nome));
```

No terminal

```
tsc index.ts --target es5
```

```
node index.js
```

Programação Orientada a Objetos em Typescript

POO

- ES6 oferece suporte a POO, porém a sintaxe de Javascript torna a POO mais complicada
- Typescript oferece uma sintaxe mais próxima de linguagens OO tradicionais, tais como Java e C#
  - Classes, herança, interfaces, sobrescrita de métodos...

Classes e herança (ooExemplo1.ts)_

```
class Animal
{
    name: string
    age: number
    breed: string
    constructor(name: string, age: number, breed: string)
    {
        this.name = name
        this.age = age
        this.breed = breed
    }
    makeSound_(sound: string): void
    {
        console.log(sound)
        console.log(sound)
        console.log(sound)
    }
}
```

Classes e herança

```
class Dog extends Animal
{
    playsFetch: boolean
    constructor(name: string, age: number, breed: string, playsFetch: boolean)
    {
         super(name, age, breed) // call parent constructor
         this.playsFetch = playsFetch
    }
    makeSound(): void
    {
        super.makeSound_('woof woof')
    }
    getAgeInHumanYears(): number
    {
        return this.age * 7
    }
}
```

Classes e herança

```
class Cat extends Animal
{
    constructor(name: string, age: number, breed: string)
    {
        super(name, age, breed)
    }
    makeSound(): void
    {
        super.makeSound_('meow meow')
    }
}

var dog1 = new Dog("Napoleao", 4, "Pastor Alemao", true)
var cat1 = new Cat("Felix", 2, "Angora")
dog1.makeSound()
cat1.makeSound()
```

Controle de acesso

- Typescript oferece modificadores public, private e protect
- Segundo boas práticas de programação, atributos (propriedades) de classes, que são representados por variáveis de instância, devem ser privados
  - Operações de escrita e leitura dessas variáveis devem ser feitas via métodos chamados ***getters*** e ***setters***

getters / setters

- Typescript provê um mecanismo que permite disparar os métodos getters e setters sem que se tenha que explicitamente chamar esses métodos
  - Basta referenciar a propriedade que o método é executado

getters / setters (ooExemplo2.ts)

```
class Dog
{
    private _name: string // underscore é convenção para var privada
    get name(): string
    {
       return this._name
    }
    set name(name: string)
    {
        if(!name || name.length > 20) {
            throw new Error('Name invalid')
        }
        else {
            this._name = name
        }
    }
}
```

getters / setters

```
class PetStore
{
    private _dogs: Array<Dog>
    constructor()
    {
        this._dogs = [new Dog(), new Dog()]
        this._dogs[0].name = 'Fido' // chama o 'set'
        this._dogs[1].name = 'Leopoldo'  // chama o 'set'
    }
    printAllDogNames(): void
    {
        this._dogs.forEach(dog => {
            console.log(dog.name) // chama o 'get'
        })
    }
}

var ps = new PetStore()
ps.printAllDogNames()
```

Modificador *protected*

- O modificador *protected* especifica que uma variável ou método só pode ser acessado por classes filhas, além da própria classe
- A classe Animal que criamos no primeiro exemplo tem o método makeSound. Como nem todo animal faz barulho, temos que criar um mecanismo que impeça que esse método seja utilizado de forma errada
  - Vamos utilizar o modificador *protected* e ver o resultado

Usando protected

```
class Animal
{
    protected makeSound_(sound: string): void
    {
        console.log(sound)
        console.log(sound)
        console.log(sound)
    }
}

class Dog extends Animal
{
    makeSound(): void
    {
        super.makeSound_('woof woof')
    }
}
```

Usando protected

```
class PetStore
{
    makeSomeSounds(): void
    {
        let dog = new Dog()
        dog.makeSound() // => 'woof woof' 'woof woof' 'woof woof'
        let animal = new Animal()
        animal.makeSound_() // => NÃO PERMITIDO
    }
}
```

Modificador *static*

- Permite que uma propriedade ou um método de uma classe sejam usados sem a necessidade de criação de instâncias

Usando static

```
class Dog
{
    static species = 'Labrador'
    age = 10
}
class PetStore
{
    printSpecies(): void
    {
        console.log(Dog.species) // =>  'Labrador'
        console.log(Dog.age) // => indefinido
    }
}
```

Modificador *readonly*

- Impede que o valor de uma propriedade seja modificado
  - Funcionamento parecido com const, mas não é a mesma coisa

Usando readonly

```
class Dog
{
    static readonly species = 'Labrador'
}
class PetStore
{
    printSpecies(): void
    {
        console.log(Dog.species) // => 'Labrador'
        Dog.species = 'Chiuaua' // => NÃO PERMITIDO
    }
}
```

Classes abstratas

- Classe concreta
  - Permite que sejam produzidas instâncias (através do operador new)
- Classe abstrata
  - Não permite instanciação, ou seja, não é possível usar o operador new com classes abstratas
  - São implementadas com o objetivo de serem estendidas por subclasses concretas
  - Contém atributos e comportamentos comuns a suas subclasses

Classes abstratas

- **Classes abstratas** podem conter ou não métodos abstratos, mas se uma classe possui **ao menos um método abstrato**, esta **deve** ser declarada como **abstrata**
- Se uma classe é subclasse de uma classe abstrata, então ela é obrigada a implementar todos os seus métodos abstratos
- Método abstrato tem a função de **forçar um comportamento** nas subclasses

Exemplo

- A classe Animal que definimos anteriormente não deve ser concreta, pois sempre criamos instâncias de Dog ou Cat
- Suponha agora que queremos que todas as subclasses de Animal sejam capazes de calcular a idade correspondente à idade humana
  - Ou seja, queremos forçar um comportamento nas subclasses
  - Para fazer isso, vamos declarar um método abastrato na classe Animal

Exemplo

- Dizem que deve-se multiplicar a idade do cão por 7. Assim, um cão com 5 anos equivaleria a um humano com 35 anos
- Suponha que esse cálculo varie de acordo com a espécie do animal
  - Cão: 7 anos
  - Gato: 6 anos
- Vamos declarar o método abstrato getRelativeAge() para forçar esse comportamento

Exemplo: classe e método abstrato

```
abstract class Animal
{
    private _name: string
    private _age: number
    constructor(name: string, age: number)
    {
        this._name = name
        this._age = age
    }
    get name(): string
    {
        return this._name
    }
    get age(): number
    {
        return this._age
    }
    abstract getRelativeAge(): number;   // Método abstrato não tem corpo
}
```

Exemplo: classe e método abstrato

```
class Dog extends Animal
{
    constructor(name: string, age: number)
    {
         super(name, age) // call parent constructor
    }
    getRelativeAge(): number
    {
        return this.age * 7
    }
}
```

Exemplo: classe e método abstrato

```
class Cat extends Animal
{
    constructor(name: string, age: number)
    {
        super(name, age)
    }
    getRelativeAge(): number
    {
        return this.age * 6
    }
}

var dog1 = new Dog("Napoleao", 4)
var cat1 = new Cat("Felix", 2)
console.log('Idade relativa de ' + dog1.name + ': ' + dog1.getRelativeAge())
console.log('Idade relativa de ' + cat1.name + ': ' + cat1.getRelativeAge())
```