Learn TypeScript

Table of Contents

Learn TypeScript Fundamentals

What is TypeScript?

TypeScript é um superset tipado do JavaScript, ou seja, é como o C++ para o C, uma melhoria da linguagem, mas de forma geral continua sendo JavaScript. Já que ele compila o código para JavaScript.

> ⚠ Este superset é open source e criado pela Microsoft.

Ele possui **três partes**:

- Linguagem (**Language**) --> a linguagem de programação em si
- Linguagem Servidor (**Language Server**) --> serve para ver em tempo construção validações de tipos, lint, hint, etc
- **Compiler** (Compilador) --> é o que torna o TypeScript para JavaScript limpo

Why?

Em um exemplo simples, se tivermos que adicionar por exemplo uma nova milf ao sistema com a função:

```
let milfStorage;

function addNewMilf(newMilf) {
    milfStorage.push(newMilf);
}
```

Logo de cara temos dois grandes problemas:

- qual o tipo de `milfStorage`? O que ela guarda? O que ela pode receber?
- o parametro `newMilf` é de que tipo? ele aceita ser o que?

São perguntas que no momento que estamos desenvolvendo talvez não pensamos, mas quando outra pessoa for olhar o código ou até mesmo você quando estiver para usar ou desenvolvendo mais esse sistema vai se deparar com um grande problema de não saber muito probvavelmente os tipos daquele objeto, ou mesmo, qual foi o erro que deu para não salvar os dados.

Logo esse superset, nasce principalmente para salvar vidas dos desenvolvedores, já que se usarmos **TS** teremos algo assim:

```
let milfStorage: String[] = new String();

function addNewMilf(newMilf: string) : void {
    milfStorage.push(newMilf);
}
```

Com isso o que temos é uma melhoria não só na legibilidade, mas também em tempo de build, já que se algo sintaticamente estiver errado a build vai parar e o language server do TS vai nos avisar qual foi o erro.

JavaScript Primer

Tipos Primitivos

Nome	Descrição
number	Usado para representar valores numéricos. JavaScript não diferencia entre inteiros e valores de ponto flutuante.
string	Usado para representar dados de texto.
boolean	Pode ter valores `true` e `false`.
symbol	Usado para representar valores constantes únicos, como chaves em coleções.
null	Pode receber apenas o valor `null` e é usado para indicar uma referência inexistente ou inválida.
undefined	Usado quando uma variável foi definida mas não foi atribuído um valor.
object	Usado para representar valores compostos, formados a partir de propriedades e valores individuais.

Comparação de Igualdade Abstrata (==)

A comparação `x == y` produz `true` ou `false` seguindo estas regras:

1. Se `Type(x)` é o mesmo que `Type(y)`, então:
    - Retorna o resultado da Comparação de Igualdade Estrita `x === y`

2. Se `x` é `null` e `y` é `undefined`, retorna `true`
3. Se `x` é `undefined` e `y` é `null`, retorna `true`
4. Se `Type(x)` é Number e `Type(y)` é String, retorna o resultado da comparação `x == ToNumber(y)`
5. Se `Type(x)` é String e `Type(y)` é Number, retorna o resultado da comparação `ToNumber(x) == y`
6. Se `Type(x)` é Boolean, retorna o resultado da comparação `ToNumber(x) == y`
7. Se `Type(y)` é Boolean, retorna o resultado da comparação `x == ToNumber(y)`
8. Se `Type(x)` é String, Number ou Symbol e `Type(y)` é Object, retorna o resultado da comparação `x == ToPrimitive(y)`
9. Se `Type(x)` é Object e `Type(y)` é String, Number ou Symbol, retorna o resultado da comparação `ToPrimitive(x) == y`
10. Retorna `false`

Comparação de Igualdade Estrita (===)

A comparação `x === y` produz `true` ou `false` seguindo estas regras:

1. Se `Type(x)` é diferente de `Type(y)`, retorna `false`
2. Se `Type(x)` é Number, então:
   - Se `x` é NaN, retorna `false`
   - Se `y` é NaN, retorna `false`
   - Se `x` é o mesmo valor Number que `y`, retorna `true`
   - Se `x` é +0 e `y` é -0, retorna `true`
   - Se `x` é -0 e `y` é +0, retorna `true`
   - Retorna `false`
3. Retorna `SameValueNonNumber(x, y)`

> **Nota:** Este algoritmo difere do Algoritmo SameValue no tratamento de zeros com sinal e NaNs.

Desviando da coerção de tipo

```
let hatPrice = 100;
console.log(`Hat price: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Boots price: ${bootsPrice}`);
if (hatPrice === bootsPrice) {
    console.log("Prices are the same");
} else {
    console.log("Prices are different");
}
let totalPrice = Number(hatPrice) + Number(bootsPrice);
console.log(`Total Price: ${totalPrice}`);
let myVariable = "Adam";
console.log(`Type: ${typeof myVariable}`);
myVariable = 100;
console.log(`Type: ${typeof myVariable}`);
```

Trabalhando com funções

```
let hatPrice = 100;
console.log(`Hat price: ${hatPrice}`);
let bootsPrice = "100";
console.log(`Boots price: ${bootsPrice}`);
function sumPrices(first, second, third) {
    return first + second + third;
}
let totalPrice = sumPrices(hatPrice, bootsPrice);
console.log(`Total Price: ${totalPrice}`)
```

Como por padrão vai fazer com que a coersão ou alguma especificidade do tipo, aconteça ou seja que com tipos como string, boolean, object a função vai ser invocada

⚠ O tipo do parametro da função é determinada pelo valor que é usado na invocação dela

Strict Mode

⚠ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

É uma forma de escrever JavaScript diferente com suas próprias regras e claro com regras do ECMA, mas tento suas diferenças

- Uma das questões é que ele elimina os erros silênciosos por throw errors
- Proibe algumas sintaxes como as que vão ser definidas nas versões futuras do ECMAScript

Entendo this in Arrow Functions

Arrow Functions são diferentes das funções comuns:

- Não possuem os valores do seu *this* e herança

```
let myObject = {
    greeting: "Hi, there",
    getWriter() {
        return (message) => console.log(`${this.greeting}, ${message}`);
    }
}
greeting = "Hello";
let writer = myObject.getWriter();
writer("It is raining today");
let standAlone = myObject.getWriter;
```

```
let standAloneWriter = standAlone();
standAloneWriter("It is sunny today");
```

Checking Prototype Types

o operador `instanceof` é usado para determinar qualquer construtor de um prototype é uma parte da corrente para um objeto especifico:

Para usar esse operador fazemos assim:

```
object instanceof anotherObject
```

Exemplo:

```
console.log(`hat and TaxedProduct: ${ hat instanceof TaxedProduct}`);
```

Classes

Classes no JavaScript foi adicionado para ser uma transição para mais de desenvolvedores de outras linguagens como Java e C#

Por detras dos panos, classes no JavaScript são implementadas usando prototypes, o que significa que temos diferenças na implementação em comparação com C# ou Java

> ⚠ Classes são definidas com a keyword: `class`

Exemplo:

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }
    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}
let hat = new Product("Hat", 100);
let boots = new Product("Boots", 100);
console.log(hat.toString());
console.log(boots.toString())
```

Inheritance in Classes

1. Com a keyword `extends` definimos qual vai ser a classe que vai ser a pai para herdarmos:

```
class TaxedProduct extends Product {
```

2. Com o keyword `super()` fazemos o uso do construtor da classe que estamos herdando:

```
constructor(name, price, taxRate = 1.2) {
    super(name, price);
    this.taxRate = taxRate;
}
```

Exemplo:

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

class TaxedProduct extends Product {
    constructor(name, price, taxRate = 1.2) {
        super(name, price);
        this.taxRate = taxRate;
    }
```

```
    getPriceIncTax() {
        return Number(this.price) * this.taxRate;
    }

    toString() {
        let chainResult = super.toString();
        return `${chainResult}, Tax: ${this.getPriceIncTax()}`;
    }
}

let hat = new TaxedProduct("Hat", 100);
let boots = new TaxedProduct("Boots", 100, 1.3);

console.log(hat.toString());
console.log(boots.toString());
```

Static Methods

Apenas aplicamos a keyword `static`:

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }
    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

class TaxedProduct extends Product {
    constructor(name, price, taxRate = 1.2) {
        super(name, price);
        this.taxRate = taxRate;
    }
    getPriceIncTax() {
        return Number(this.price) * this.taxRate;
    }
    toString() {
        let chainResult = super.toString();
        return `${chainResult}, Tax: ${this.getPriceIncTax()}`;
    }
    static process(...products) {
        products.forEach(p => console.log(p.toString()));
    }
}

TaxedProduct.process(new TaxedProduct("Hat", 100, 1.2),
    new TaxedProduct("Boots", 100));
```

Iterators and Generators

Iterators

Iterators são objetos que retornam uma sequencia de valores e eles são usados com coleções

Um iterator define a função chamada next que retorna um objeto com valores e propriedades feitas:

- O valor da propriedade retorna o próximo valor na sequencia
- A propriedade feita é setada para true quando a sequencia é completa

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }
    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

function createProductIterator() {
    const hat = new Product("Hat", 100);
    const boots = new Product("Boots", 100);
    const umbrella = new Product("Umbrella", 23);
    let lastVal;

    return {
        next() {
            switch (lastVal) {
                case undefined:
                    lastVal = hat;
                    return { value: hat, done: false };
                case hat:
```

```
                    lastVal = boots;
                    return { value: boots, done: false };
                case boots:
                    lastVal = umbrella;
                    return { value: umbrella, done: false };
                case umbrella:
                    return { value: undefined, done: true };
            }
        }
    }
}

let iterator = createProductIterator();
let result = iterator.next();

while (!result.done) {
    console.log(result.value.toString());
    result = iterator.next();
}
```

Generator

É uma forma mais fácil de criar porque usamos a keyword `yield` e a ideia é que escrever iterators pode ser não muito legal porque o código tem que se manter o estado dos dados sendo carregado da atual posição em sequencia por cada vez na função next

Funções Generator são denotadas com asterisco:

```
function* createProductIterator() {
    yield new Product("Hat", 100);
    yield new Product("Boots", 100);
    yield new Product("Umbrella", 23);
}
```

Podemos usar Generator com Spread Operator:

```
[...createProductIterator()].forEach(p => console.log(p.toString()))
```

Defining Iterable Objects

Funções standalone (funções independentes), para iterators e generators podem ser muito uteis, mas o mais o comum requisito é para um objeto para prover uma sequencia de parte de alguma funcionalidade

Exemplo:

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

class GiftPack {
    constructor(name, prod1, prod2, prod3) {
        this.name = name;
        this.prod1 = prod1;
        this.prod2 = prod2;
        this.prod3 = prod3;
    }

    getTotalPrice() {
        return [this.prod1, this.prod2, this.prod3]
            .reduce((total, p) => total + p.price, 0);
    }

    *getGenerator() {
        yield this.prod1;
        yield this.prod2;
        yield this.prod3;
    }
```

```
}

let winter = new GiftPack("winter", new Product("Hat", 100),
    new Product("Boots", 80), new Product("Gloves", 23));

console.log(`Total price: ${winter.getTotalPrice()}`);

[...winter.getGenerator()].forEach(p => console.log(`Product: ${p}`))
```

Definindo Default Iterator

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }
    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}
class GiftPack {
    constructor(name, prod1, prod2, prod3) {
        this.name = name;
        this.prod1 = prod1;
        this.prod2 = prod2;
        this.prod3 = prod3;
    }
    getTotalPrice() {
        return [this.prod1, this.prod2, this.prod3]
            .reduce((total, p) => total + p.price, 0);
    }
    *[Symbol.iterator]() {
        yield this.prod1;
        yield this.prod2;
        yield this.prod3;
    }
}
```

```
let winter = new GiftPack("winter", new Product("Hat", 100),
    new Product("Boots", 80), new Product("Gloves", 23));
console.log(`Total price: ${ winter.getTotalPrice() }`);
[...winter].forEach(p => console.log(`Product: ${ p }`))
```

⚠ A propriedade `Symbol.iterator` é usada oara denotada o iterator padrão para um objeto

Assim que consigo usar desse jeito: `[...winter].forEach(p => console.log(Product: p))\> Semprecisarfazerassim :>\[...winter.getGenerator()].forEach(p =>));`

Collections

De forma geral, o comum dentro das collections do JavaScript usamos objects e arrays:

- Arrays: `let cards = ["card1", "card2"];`
- Objects: `let user = {name: "Vampeta"};`

Exemplo:

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    toString() {
        return `toString: Name: ${this.name}, Price: ${this.price}`;
    }
}

let data = {
    hat: new Product("Hat", 100)
}

data.boots = new Product("Boots", 100);
Object.keys(data).forEach(key => console.log(data[key].toString()))
```

Armazenando dados pela chave com Object

⚠ `Object.keys(object)` usado para retornar um array contendo os nomes definidos apropriadamente pelo objeto `Object.values(object)` esse método é usado para retornar um array com todos os valores do objeto

Assim conseguimos pegar a chave do objeto sem saber qual ela é exatamente e acessar o valor da propriedade do objeto pela chave e assim armazenar deste modo:

```
Object.keys(data).forEach(key => console.log(data[key].toString()))
```

Armazenando dados por um Map

```
class Product {
    constructor (name, price) {
        this.name = name;
        this.price = price;
    }

    toString () {
        return `toString: Name ${this.name} ; Price: ${this.price}`
    }
}

let data = new Map();

data.set("hat", new Product("Hat", 100));
data.set("boots", new Product("Boots", 100));

[...data.keys()].forEach(key => console.log(data.get(key).toString()));
```

⚠ Confira a API do Map: clique aqui (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Map)

Alguns métodos importantes:

Método	Função	Exemplo
`set()`	Armazena um valor com a chave especificada.	`map.set('nome', 'Gustavo')`
`get()`	Recupera o valor armazenado com a chave especificada.	`map.get('nome')`
`keys()`	Retorna um iterador para as chaves no Map.	`for (let k of map.keys())`
`values()`	Retorna um iterador para os valores no Map.	`for (let v of map.values())`
`entries()`	Retorna um iterador para os pares chave/valor como arrays.	`for (let e of map.entries())`

Vantagens do Map

- Não contem nenhuma chave por padrão, só tem o que foi definido
- As chaves podem ser qualquer valor, de qualquer tipo, desde built in como objetos complexos
- As cheaves dentro do map são ordenados de forma simples, o objeto Map itera suas entradas chaves e valor na ordem em que foram inseridas
- Podemos pegar o número de items dentro do Map com a propriedade `size`
- Diretamente iterável
- Funciona melhor em cenarios em que precisamos armazenar e remover por chave-valor
- Não tem suporte para serialização ou analise sintática, mas pode ser implementado: veja mais na discussion do StackOverflow

(https://stackoverflow.com/questions/29085197/how-do-you-json-stringify-an-es6-map)

Usando Symbols para chaves do Map

A principal vantagem de usar um Map é que qualquer valor pode ser uma chave (*key*), incluindo **Symbol (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)**

Cada valor de um Symbol é único e imutavel e idealmente escolhido para um identificador de objetos

```
class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}
class Supplier {
    constructor(name, productids) {
        this.name = name;
        this.productids = productids;
    }
}
let acmeProducts = [new Product("Hat", 100), new Product("Boots", 100)];
let zoomProducts = [new Product("Hat", 100), new Product("Boots", 100)];
let products = new Map();

[...acmeProducts, ...zoomProducts].forEach(p => products.set(p.id, p));

let suppliers = new Map();

suppliers.set("acme", new Supplier("Acme Co", acmeProducts.map(p =>
p.id)));
suppliers.set("zoom", new Supplier("Zoom Shoes", zoomProducts.map(p =>
p.id)));
```

```
suppliers.get("acme").productids.forEach(id =>
        console.log(`Name: ${products.get(id).name}`));
```

O benefício de usar Symbol `this.id = Symbol();` é que não é possivel termos duas keys com valores iguais, ou seja, não teremos uma colisão de dados

Armazenando dados pelo index

Um dos tipos de listas que JavaScript provê é o Set, que armazenar dados pelop indice mas tem performance otimizada e mais útil, armazenando somente valores únicos:

```
class Product {
    constructor(name, price) {
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}
let product = new Product("Hat", 100);
let productArray = [];
let productSet = new Set();
for (let i = 0; i < 5; i++) {
    productArray.push(product);
    productSet.add(product);
}
console.log(`Array length: ${productArray.length}`);
console.log(`Set size: ${productSet.size}`);
```

Como a saída será em que o `Set size: 1` justamente porque ele gera a integridade e consistência dos dados eliminando duplicatas, diferente de arrays que a saída será `Array length: 5` já que ele permite duplicatas

Alguns métodos uteis:

Método	Função	Exemplo
`add()`	Adiciona um valor ao Set.	`set.add('banana')`
`entries()`	Retorna um iterador com os itens do Set como pares [valor, valor].	`for (let e of set.entries())`
`has()`	Retorna `true` se o Set contém o valor especificado.	`set.has('banana')`
`forEach()`	Executa uma função para cada valor no Set.	`set.forEach(v => console.log(v))`

⚠ Confira a documentação da API do Set: clique aqui (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Set)

Defining Static Properties and Method

As propriedades e metodos definidos na função construturora são normalmente chamados de **estáticos**, significa que são acessados atraves do constructor e por objetos individuais criados por esse constructor (que é o oposto para propriedades de instância, que são acessadas atravez de um objeto)

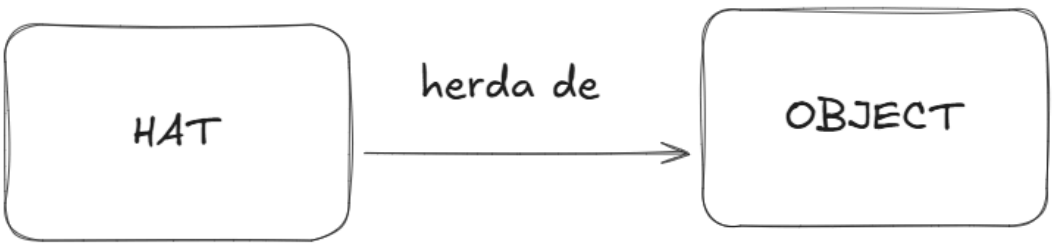
The `Object.setPrototypeOf` e `Object.getPrototypeOf` esses métodos são bons exemplos de statics methods

Inheritance

No JavaScript os objetos tem links para outros objetos, conhecidos como **prototypes**, cada um deles tem suas propriedades e métodos

Prototypes são objetos, e tem suas próprias propriedades, objetos formam um corrente de herança (*inheritance chain*), que segue funcionalidades complexas para serem definidas uma vez e usada consistentemente

```
let hat = {
    name: "Hat",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
};
console.log(`Hat: ${hat.price}, ${hat.getPriceIncTax() }`);
console.log(`toString: ${hat.toString()}`);
```



A imagem representa o objeto e o prototipo

Inspecionando e Modificando o prototype de um objeto

Object é um prototype para muitos objetos, mas isso também prove alguns metodos que são usados diretamente, pela herança e que também será usado para obter informação sobre o prototype

Alguns métodos:

Nome do Método	Descrição da Função
`getPrototypeOf`	Retorna o **protótipo** (o objeto ou `null`) do objeto especificado.
`setPrototypeOf`	**Altera o protótipo** (o objeto interno `[[Prototype]]`) de um objeto especificado para outro objeto ou `null`.
`getOwnPropertyNames`	Retorna um *array* contendo os **nomes de todas as propriedades próprias** (não herdadas) e enumeráveis ou não enumeráveis de um objeto.

Exemplo:

```
let hat = {
    name: "Hat",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
};

let boots = {
    name: "Boots",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
};

let hatPrototype = Object.getPrototypeOf(hat);
console.log(`Hat Prototype: ${hatPrototype}`);
let bootsPrototype = Object.getPrototypeOf(boots);

console.log(`Boots Prototype: ${bootsPrototype}`)
```

```
console.log(`Hat: ${hat.price}, ${hat.getPriceIncTax() }`);
console.log(`toString: ${hat.toString()}`);
```

Os objetos derivam do mesmo prototype:

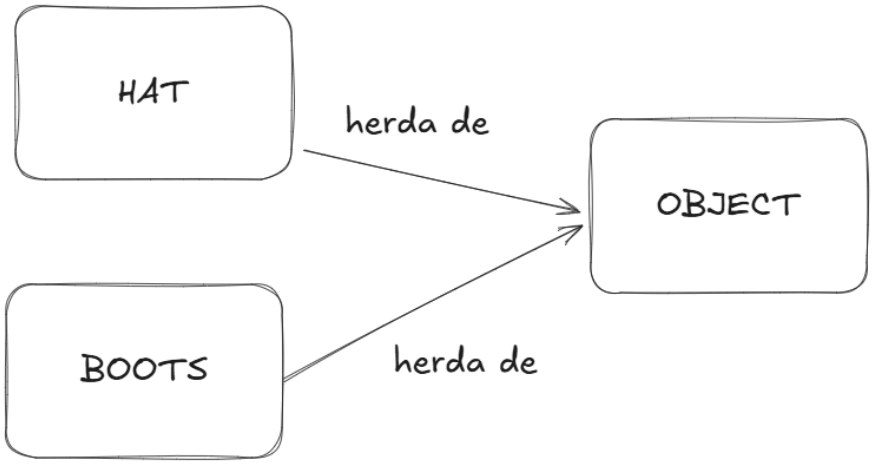


image.png

Podemos definir novas propriedades e métodos para os prototypes, enovos valores podem ser adicionados para propriedades existentes:

```
let hat = {
    name: "Hat",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
};
let boots = {
    name: "Boots",
    price: 100,
    getPriceIncTax() {
        return Number(this.price) * 1.2;
    }
}
```

```
let hatPrototype = Object.getPrototypeOf(hat);
hatPrototype.toString = function() {
    return `toString: Name: ${this.name}, Price: ${this.price}`;
}
console.log(hat.toString());
console.log(boots.toString())
```

Chaining Constructor Functions

Usando o método `setPrototypeOf` para criar uma encadeamento de prototypes customizados é fácil mas definir isso corretamente assim como os funções construtoras dá trabalho

1. Vai fazer a chamar o metodo que permite criar um novo objeto para ser passado para o próximo construtor com esse valor:

> ⚠ Product.call(this,name,price);

2. Agora seria linkar os dois prototypes:

> ⚠ Object.setPrototypeOf(TaxedProduct.prototype, Product.prototype)

Diagrama mostrando as heranças que acontecem:

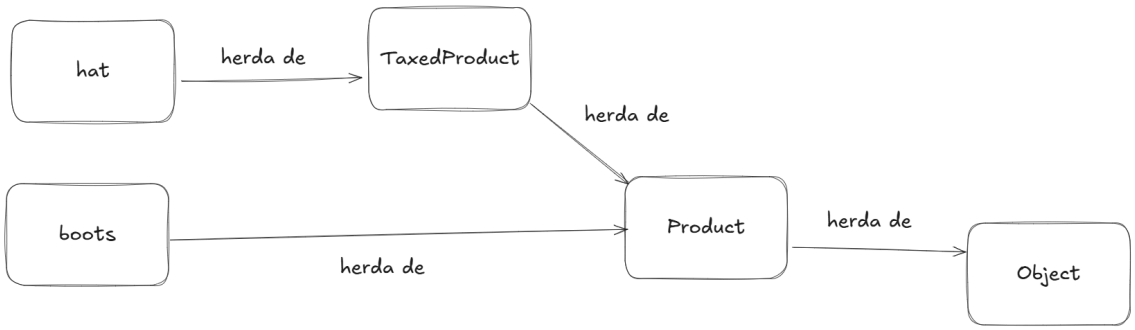


image_2.png

Modules

Como as aplicações vão crescendo e se tornando inviaveis para serem feitas em um só arquivo precisamos então de módulos, ou seja, organizar o código em diferentes arquivos/pastas

> ℹ NodeJS só suporta até o momento standard modules como uma feature experimental

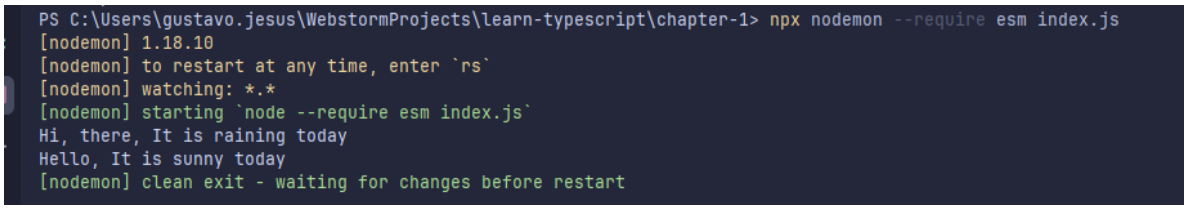
1. Instale o package:

```
npm install esm@3.2.2
```

2. Agora é rodar:

```
npx nodemon --require esm index.ts
```

3. A saída será essa:



saida do terminal usando npx nodemon --require esm

Criando um módulo JavaScript

Primeiro temos que entender que cada arquivo JavaScript é um módulo próprio

Ou seja, `app.js` e `user-controller.js` são dois módulos separados que não conversam:

two modules that they dont talk

Exemplo

Criamos um arquivo `tax.js` na pasta:

```
export default function (price) {
    return Number(price) * 1.2;
}
```

A função definida no arquivo recebe um preço e aplica 20% de taxa. A função é simples e a exportação se dá pelas keywords que são importantes: `export defaul`

- O `export` é usado para denotar que features que poderão ser usadas fora do módulo
- Por padrão o conteúdo de um arquivo JavaScript são privados e devem ser compartilhados de forma explicita usando a keyword `export` antes deles serem usados no resto da aplicação
- Agora a outra parte o `default` é usado para especificar que somente será exportado aquela função no arquivo `tax.js`

Usando Módulos

Basta usarmos a keyword: `import calcTax from "./tax";`

```
import calcTax from "./tax";

class Product {
    constructor(name, price) {
```

```
        this.id = Symbol();
        this.name = name;
        this.price = price;
    }
}

let product = new Product("Hat", 100);
let taxedPrice = calcTax(product.price);

console.log(`Name: ${ product.name }, Taxed Price: ${taxedPrice}`);
```

> ⚠ Não precisamos passar a extensão do arquivo se ele for `js` (o mesmo acontece com todas as outras ramificações que temos como `.ts` ou `.jsx`), basta passarmos o caminho do arquivo

Entendendo localização de módulos

Quando temos módulos definidos no **próprio** projeto preciamos passar o path relativo:

- `import calcTax from "./tax";`

Quando o módulo é **externo** não precisamos:

- `import React, { Component } from "react";`

> ⚠ Isso acontece porque a ferramenta de build vai buscar em `node_modules` o módulo a ser importado

Exportando feature nomeadas de um módulo:

Tipo de Exportação	Código de Exportação	Código de Importação	Observação
Exportação nomeada	`export function calculateTax(price) { ... }`	`import { calculateTax } from "./tax";`	Usa chaves `{}` para importar pelo nome definido
Exportação default	`export default function calcTaxandSum(...) { ... }`	`import calcTaxAndSum from "./tax";`	Pode ser importado com qualquer nome, sem `{}`
Exportação combinada	`export default ...` + `export function ...`	`import calcTaxAndSum, { calculateTax } from "./tax";`	Combina exportação default e nomeada
Múltiplas nomeadas	`export function printDetails(...)` `export function applyDiscount(...)`	`import { printDetails, applyDiscount } from "./utils";`	Permite agrupar várias funções relacionadas no mesmo módulo

Using TypeScript Compiler

> ⚠ https://github.com/mrpunkdasilva/learn-typescript/blob/9ef0a68b44f22c7a7be5944a44f15e51504ac838/tools (https://github.com/mrpunkdasilva/learn-typescript/blob/9ef0a68b44f22c7a7be5944a44f15e51504ac838/tools)

Para instalar o TypeScript:

```
npm i -D typescript
```

Precisamos ter o arquivo de configuração do TypeScript criado na pasta raiz do projeto:

- `tsconfig.json`

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

- `package.json`

```
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
```

```
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^2.1.2",
    "typescript": "^3.5.1"
  }
}
```

Project Structure

Item	Função
dist	Contém o código compilado gerado pelo compilador.
node_modules	Armazena os pacotes necessários para a aplicação e ferramentas.
src	Contém os arquivos de código-fonte que serão compilados pelo TypeScript.
package.json	Lista as dependências principais do projeto e metadados.
package-lock.json	Registra todas as dependências exatas e suas versões.
tsconfig.json	Define as configurações do compilador TypeScript.

Entendo o arquivo de configuração do TypeScript Compiler

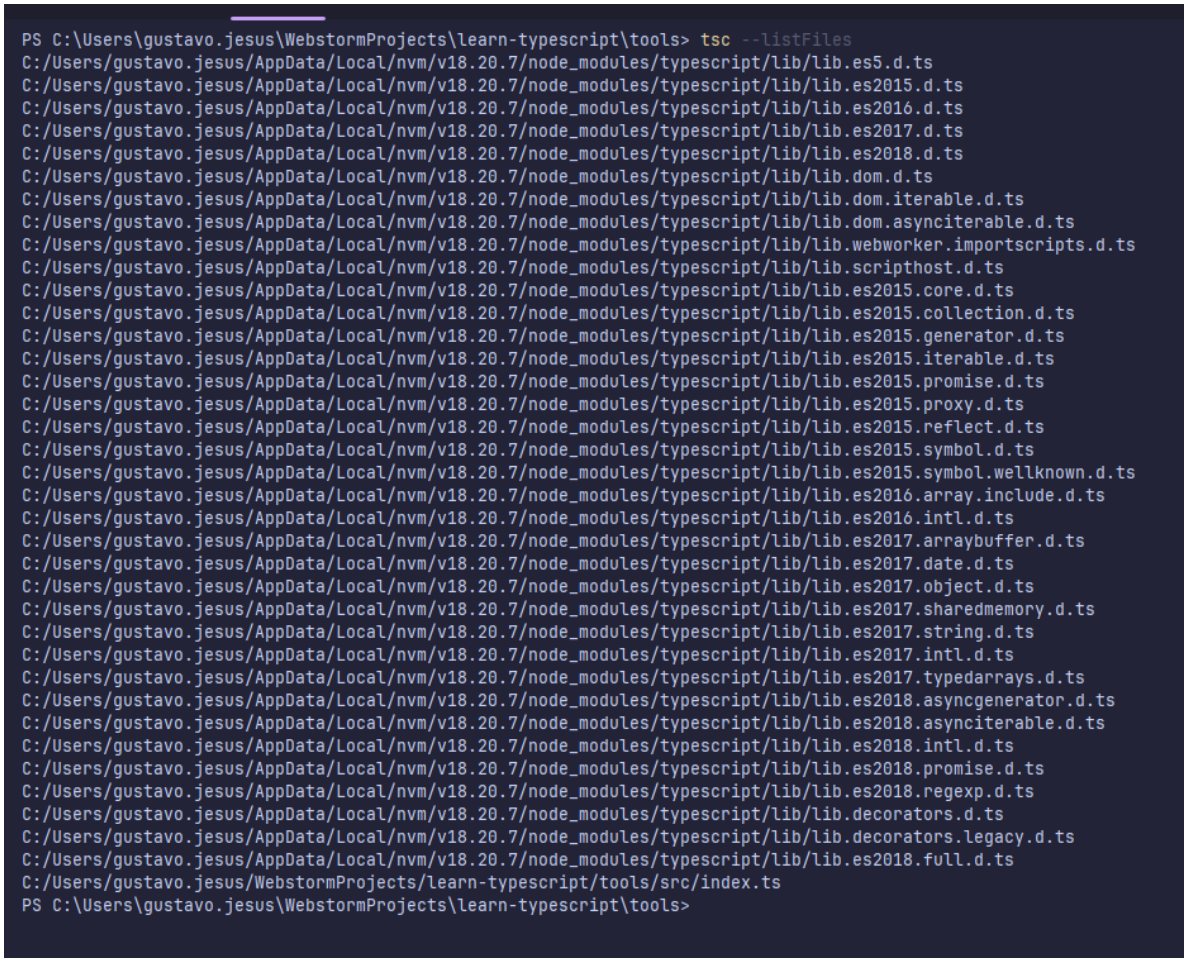
O TSC (*TypeScript Compiler*) possui diversas configurações possiveis a serem feitas e todas são feitas no arquivo de configuração que temos (`tsconfig.json`)

As configs básicas que temos são:

Configuração	Função
compilerOptions	Agrupa as opções e configurações que o compilador irá usar.
files	Especifica arquivos exatos a serem compilados (sobrescreve busca padrão).
include	Seleciona arquivos para compilação por padrão/padrão de busca.
exclude	Exclui arquivos da compilação por padrão/padrão de busca.
compileOnSave	Quando `true`, pede ao editor para compilar ao salvar (nem todos suportam).

Podemos exibir a lista de arquivos para compilação que podemos usar:

```
tsc --listFiles
```



tsc list files

Compilando código TypeScript

Para compilarmos os arquivos TS usamos o comando dentro da pasta raiz do projeto:

> ⚠ tsc

Quando o compilador roda, a saida ira ser gerada somente quando não existir erros detectados no código JS quando a config de `noEmitOnError` estiver `true`:

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
```

```
    "noEmitOnError": true
  }
}
```

Rodando com watch

Com a flag `watch` ao rodarmos temos o beneficio da recompilação automatica ao termos mudado o código

Execução automatica do código depois da compilação

O modo de compilação assistida ou compiler's watch mode não executa o código compilado

Então para resolver isso podemos usar o pacote open source do `tsc-watch`

```
npx tsc-watch --onsuccess "node dist/index.js
```

Para não precisar ficar digitando isso, podemos adicionar um script para ser rodado com o npm:

```
{
  "name": "tools",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onsuccess \"node dist/index.js\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "tsc-watch": "^2.1.2",
    "typescript": "^3.5.1"
  }
```

Logo basta rodar apenas:

⚠ npm start

Compiling TS with TSC

O **TSC** (TypeScript Compiler) é o compilador do TypeScript.

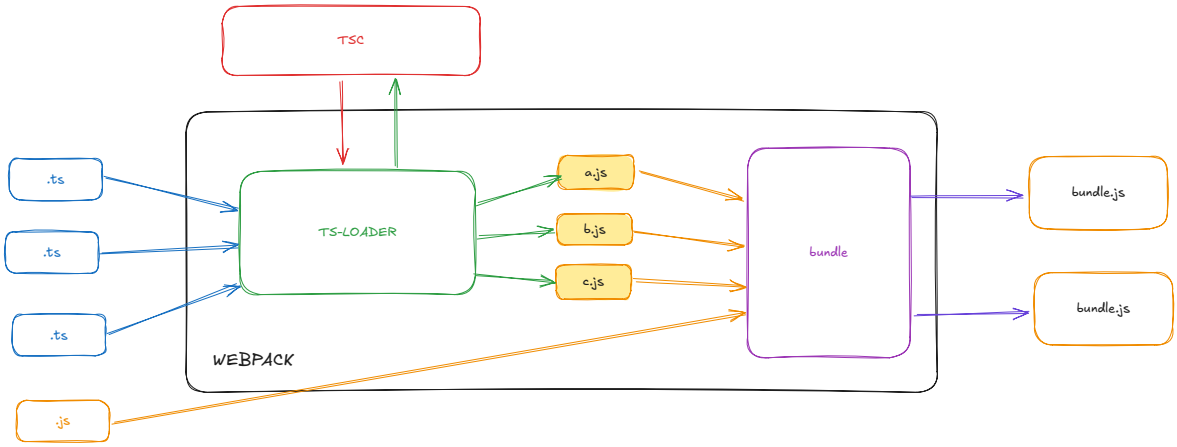


Figura 1: Diagramação de como o TypeScript funciona

Anatomia do Projeto

- `package.json` - vai ser onde as dependencias do projeto ficaram
- `tsconfig.json` - vai ser onde ficaram as configuração do compilador do TS
- `src/index.ts` - será onde vai ficar o programa

package.json

```
{
  "name": "welcome-to-ts",
  "license": "NOLICENSE",
  "devDependencies": {
    "typescript": "^5.2.0"
  },
  "scripts": {
    "dev": "tsc --watch --preserveWatchOutput"
  }
}
```

- Para compilarmos usamos o `tsc` como visto no `scripts` e
  - usando a flag `--watch --preserveWatchOutput` servira para ficar observando (ou seja, assim que houver uma atualização o compilador irá rebuildar automaticamente) e para preservar a saida de TS -> JS

tsconfig.json

```
{
  "compilerOptions": {
    "outDir": "dist",
    "target": "ES2015",
    "moduleResolution": "node",
    "module": "commonjs"
  },
  "include": ["src"]
}
```

Running

- Com o arquivo:

```
/**
 * Create a promise that resolves after some time
 * @param n number of milliseconds before promise resolves
 */
function timeout(n: number) {
  return new Promise((res) => setTimeout(res, n));
}

/**
 * Add two numbers
 * @param a first number
 * @param b second
 */
export async function addNumbers(a: number, b: number) {
  await timeout(500);
```

```
  return a + b;
}

class Foo {
  static #bar = 3;
  static getValue() {
    return Foo.#bar;
  }
}

//== Run the program ==//
(async () => {
  console.log(await addNumbers(Foo.getValue(), 4));
})();
```

- Agora vamos rodar, para isso use: `npm run dev` ativar o script para ficar observando o arquivo para automatico rebuild a cada alteração
- Então bastar irmos agora para o terminal e digitar `node dist/index.ts` que é a pasta e o arquivo que o compilador gerou ao compilarmos e assim o node vai executar esse arquivo JavaScript

> ⚠ Para compilar chamando diretamente o compilador (TSC) usamos: `[gerenciador de pacote] tsc`
>
> - `npm tsc`
> - `yarn tsc`

Using target version

Como o TS é um superset, é como pegarmos alguma versão do JS e termos novas features, ou seja. Com o TS conseguimos usar diversas versões target (alvo) do JS para trabalharmos

Valor	Descrição
es3	Terceira edição (1999). Base da linguagem. É o valor padrão se não definido.
es5	Quinta edição (2009). Foco em consistência. *(Não houve ES4)*.
es6	Sexta edição. Introduziu classes, módulos, arrow functions e promises.
es2015	Equivalente ao ES6.
es2016	Sétima edição. Incluiu `Array.includes` e operador de exponenciação.
es2017	Oitava edição. Novos recursos para objetos e palavras-chave assíncronas.
es2018	Nona edição. Introduziu operadores *spread/rest* e melhorias em strings e async.
esNext	Recursos previstos para a próxima edição. Pode variar entre versões do TS.

Para mudarmos basta irmos na propertie do `tsconfig.json` de `target`:

> ⚠ "target": "es5",

```
{
    "compilerOptions": {
```

```
        "target": "es5",
        "outDir": "./dist",
        "rootDir": "./src",
        "noEmitOnError": true
    }
}
```

Modules

- **Objetivo**: dividir aplicações em múltiplos arquivos para facilitar manutenção.
- **Export/Import**:
  - `export` → disponibiliza funções/valores.
  - `import` → usa funções/valores de outro arquivo.

Exemplo:

```
// calc.ts
export function sum(...vals: number[]): number {
  return vals.reduce((total, val) => total + val);
}

// index.ts
import { sum } from "./calc";
console.log(sum(100, 200, 300)); // 600
```

Configuração no tsconfig.json

- O compilador usa a opção **`target`** e **`module`** para decidir como gerar código.
- **Node.js** suporta **CommonJS** por padrão.
- Versões mais novas (ES2015+) usam `import`/`export` diretamente, mas podem dar erro no Node sem configuração extra.

Exemplo de configuração:

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
```

```
    "noEmitOnError": true,
    "module": "commonjs"
  }
}
```

Tipos de módulo (opção module)

Valor	Descrição
none	Desativa módulos.
commonjs	Padrão do Node.js.
amd	Usado pelo loader RequireJS.
system	Usado pelo loader SystemJS.
umd	Universal, funciona em vários ambientes.
es2015/es6	Usa padrão de módulos ES2015 (import/export).
esnext	Usa recursos de módulos propostos para versões futuras.

Resolução de módulos

- **classic** → busca módulos apenas no projeto.
- **node** → busca também em `node_modules`.
- Configurado com `moduleResolution: "classic"` ou `"node"`.

⚠ Resumindo:

- Para **Node.js**, use `"module": "commonjs"`.

- Para **apps web com bundlers** (React, Angular, Vue), use `"module": "es2015"` ou `"esnext"`.

Useful compiler options

Opção	Função
allowJs	Inclui arquivos `.js` na compilação.
allowSyntheticDefaultImports	Permite importar módulos sem `default export`.
baseUrl	Define a pasta raiz para resolver módulos.
checkJs	Verifica erros em arquivos `.js`.
declaration	Gera arquivos `.d.ts` com informações de tipos.
downlevelIteration	Suporte a iteradores em versões antigas de JS.
emitDecoratorMetadata	Inclui metadados de *decorators* no JS emitido.
esModuleInterop	Facilita importação de módulos CommonJS.
experimentalDecorators	Ativa suporte a *decorators*.
forceConsistentCasingInFileNames	Garante que nomes em `import` respeitem maiúsculas/minúsculas.
importHelpers	Usa helpers para reduzir código gerado.
isolatedModules	Trata cada arquivo como módulo separado (compatível com Babel).
jsx	Define como JSX/TSX será processado.
jsxFactory	Define a função usada para criar elementos JSX.

lib	Seleciona bibliotecas padrão usadas pelo compilador.
module	Define o formato de módulos (ex.: `commonjs`, `esnext`).
moduleResolution	Define como módulos são resolvidos (`classic` ou `node`).
noEmit	Não gera saída JS, apenas verifica erros.
noImplicitAny	Impede uso implícito do tipo `any`.
noImplicitReturns	Exige que todas as funções retornem valor.
noUnusedParameters	Alerta sobre parâmetros não usados.
outDir	Define a pasta de saída dos arquivos compilados.
paths	Define caminhos personalizados para resolver módulos.
resolveJsonModule	Permite importar arquivos `.json` como módulos.
rootDir	Define a pasta raiz dos arquivos TypeScript.
skipLibCheck	Ignora checagem de arquivos de declaração para acelerar compilação.
sourceMap	Gera *source maps* para depuração.
strict	Ativa verificações estritas de tipos.
strictNullChecks	Impede `null`/`undefined` em tipos não compatíveis.

suppressExcessPropertyErrors	Evita erro quando objeto tem propriedades extras.
target	Define versão de JS gerada (ex.: `es5`, `es2018`).
typeRoots	Define pasta raiz para arquivos de declaração de tipos.
types	Lista arquivos de declaração a incluir na compilação.

Structural and Nominal Types

Static vs Dynamic

- **Estático**: vai fazer a checagem de tipo na hora da compilação ou não
- **Dinamica**: vai fazer a checagem de tipo em tempo de execução

O TS tem como seu tipo de checagem o **estático**

Ducking Typing

É um tipo de tipagem que é feito usando o teste do pato:

> ⚠ “If it looks like a duck, swims like a duck, and quacks like a duck, then it’s probably is a duck”.

Ou seja, ele vai tipar o que aquilo parece ser, é muito usado para sistemas com tipagem dinamica

Tipagem "fraca" e "forte"

No contexto do TS o caminho comum é usar a tipagem forte.

Nominal vs Structural

- O **nominal** é o sistema de tipagem que foca em **nomes**
  - Como por exemplo no código abaixo, o sistema nominal, ele vai verificar se a equivalencia de tipo de `myCar` é a mesma de `checkCar' (exemplo no Java):

```
public class Car {
  String make;
  String model;
  int make;
```

```
}

public class CarChecker {
  // takes a `Car` argument, returns a `String`
  public static String checkCar(Car car) {  }
}

Car myCar = new Car();

// TYPE CHECKING
// -------------
// Is `myCar` type-equivalent to
//     what `checkCar` wants as an argument?
CarChecker.checkCar(myCar);
```

- Agora o **Structural** foca num sistema de tipagem de estrutura ou tipo:
  - Se olharmos o exemplo abaixo, em TS, vemos que a structural irá focar se o objeto tem ou não a mesma estrutura que se quer:

```
class Car {
  make: string
  model: string
  year: number
  isElectric: boolean
}

class Truck {
  make: string
  model: string
  year: number
  towingCapacity: number
}

const vehicle = {
  make: "Honda",
  model: "Accord",
```

```
  year: 2017,
}

function printCar(car: {
  make: string
  model: string
  year: number
}) {
  console.log(`${car.make} ${car.model} (${car.year})`)
}

printCar(new Car()) // Fine
printCar(new Truck()) // Fine
printCar(vehicle) // Fine
```

Static Types

É quando definimos o tipo do valor da variavel

Esses tipos podem ser por exemplo os *built-in types* que são os tipos primitivos que temos tanto no TS quanto no JS

Tipo	Descrição
number	Representa valores numéricos.
string	Representa dados de texto.
boolean	Pode ter os valores `true` ou `false`.
symbol	Representa valores constantes únicos, usado como chaves em coleções.
null	Só pode receber o valor `null`, indicando referência inexistente ou inválid a.
undefine d	Indica que a variável foi definida mas não recebeu valor.
object	Representa valores compostos, formados por propriedades e valores.

Criando tipos estáticos com a type annotation (notação de tipos)

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}
```

Usamo o `:` e depois informamos o tipo (string, number, etc)

Mas temos que entender que existem alguns tipos de notação com base onde ela está:

No TypeScript, os **tipos** podem aparecer em diferentes lugares do código, e a notação muda conforme o contexto:

1. Parameter type

Define o tipo dos **parâmetros** de uma função.

```
function soma(a: number, b: number) {
  return a + b;
}
```

- Aqui, `a: number` e `b: number` são **parameter types**.

2. Return type

Define o tipo do **valor retornado** por uma função.

```
function saudacao(nome: string): string {
  return `Olá, ${nome}`;
}
```

- O `: string` depois dos parênteses indica o **return type** da função.

3. Variable type

Define o tipo de uma **variável**.

```
let idade: number = 25;
let ativo: boolean = true;
```

- `idade: number` e `ativo: boolean` são **variable types**.

Resumindo em tabela:

Tipo de notação	Onde aparece	Exemplo
parameter type	Nos parâmetros da função	`function soma(a: number, b: number)`
return type	Após os parênteses da função	`function saudacao(nome: string): string`
variable type	Na declaração de variáveis	`let idade: number = 25`

Usando definição implicita de tipos estáticos

```
function calculateTax(amount: number) {
    return amount * 1.2;
}
let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;
console.log(`Full amount in tax: ${taxAmount}`);
console.log(`Half share: ${halfShare}`)
```

Com isso o compilador assume que o tipo é number

> Podemos dizer ao compilador para gerar arquivos que contenha informações dos tipos por todo o codigo JS produzido. Basta editarmos o `tsconfig.json` e adicionar essa compile option: `"declaration": true`

Variables and Values

Podemos declara por inferencia:

```
let foo = 12;
foo = "12"
```

Um tipo como um conjunto de valores permitidos

Com isso o tipo por inferencia ficou como `number` logo ao tentarmos atribuirmos um valor de string a variavel `foo` será nós avisado que isto está incorreto;

- O erro será de **assinatura de tipo**, ou seja, é quando por inferencia pensa que aquela mulher no tinder é do tipo mulher, mas quano chega lá é do tipo mulher feijoada, vem com linguiça

No TS quando criamos uma constante e atribuimos um valor ele pega esse valor e assume como o tipo da variavel:

```
const humidity = 79
```

- O tipo de `humidity` é `79`, meio estranho eu sei, mas ele faz essa assumição específica aqui
- A `const` não pode ser retribuída e o tipo de `humidity` é tipo de valor imutável

Por exemplo, não poderíamos fazer isto:

- já que a variavel `temperature` não é do tipo `79`;

```
humidity = temperature;
```

Seguindo isso também serviria para as variaveis mutaveis:

```
let temperature = 79 as 79
```

Com esse `as` dizemos que o tipo é `79` e não um número:

- Ao tentarmos atribuir um valor do tipo númerico não conseguimos, já que o tipo esperado é `79`

```
temperature = 12 // Isto resultara também em erro de assinatura de tipo
```

Temos outro tipo de loucura no TS que é criarmos uma `let` em que definimos que o que ela aceita é um valor especifico:

```
let temperature = 79 as const;
```

- Isso faz com que o tipo dessa variavel seja somente do tipo `79`, que não é um number e sim `79`

Tipo implicito: any e anotações de tipo

Neste exemplo vemos o uso do `any` quando apenas declaramos uma variavel ou queremos que ela assuma um valor flexivel e que fuja do bom senso de usar uma linguagem tipada por algum motivo sem sentido e hediondo usamos o `any`;

```
// between 500 and 1000
const RANDOM_WAIT_TIME =
  Math.round(Math.random() * 500) + 500

let startTime = new Date()
let endTime
```

No caso acima o tipo `any` foi assumido de forma implicita, mas podemos usala de forma explicita com usando: type annotations ou anotação de tipo:

- Para definirmos um tipo usamos o sinal de ":" (dois pontos) e passamos o tipo;

```
let endTime: any
```

Type Casting

Para fazermos o casting podemos usar o `any`:

Neste exemplo temos a variavel `date2`:

- que não definimos um tipo com notação então ela pegaria o tipo por inferencia usando o tipo do dado que foi atribuído a ela
- que neste caso está vindo da variavel `founding` que é do tipo `Date`,
- logo quando usamos `founding as any` estamos fazendo um casting para dizer que o tipo é `any` e não `Date`

```
let founding = new Date("Jan 1, 2012")
let date1 = founding
let date2 = founding as any;
```

Podemos seguir este mesmo principio para constantes:

- o tipo da constante é 79 mas pe trocado por number

```
const humid3 = 79 as number;
```

Objects

Podemos usar os objetos que já temos no JS, mas tipando eles assim:

- Aqui definimos os tipos apenas, nada inicializado:

```
let car: {
  make: string;
  model: string;
  year: number;
};
```

> ⚠ O ponto e virgula no final de cada propriedade é opcional e depende da code base do projetoo;

- Agora veremos como seria um objeto numa função:

```
function showInfoCar(car: {
    make: string; model: string; year: number;
}) : string {
    return `${make} | ${model} | ${number}`;
}
```

- Podemos também definir que determinada propriedade é opcional, usando o operador `?` depois do nome da propriedade:

```
function showInfoCar(car: {
    make: string;
    model: string;
    year: number;
    chargeVoltage?: number;
}) : string {
    return `
```

```
    ${make}   |
    ${model}  |
    ${number} |
    ${(typeof car.chargeVoltage === "number")? "Car Voltage: " +
car.chargeVoltage : "" }
    `;
}
```

- Está notação serve também para argumentos em funções, assim podemos ou não passarmos o atributo `car` no objeto:

```
function showInfoCar(car?: {
    make: string;
    model: string;
    year: number;
    chargeVoltage?: number;
}) : string {
    if (!car) {
        console.log("Don't passed car");
        return;
    }

    return `
        ${make}   |
        ${model}  |
        ${number} |
        ${(typeof car.chargeVoltage === "number")? "Car Voltage: " +
car.chargeVoltage : "" }
    `;
}
```

Assinaturas Indexadas

Esses tipo de estrutura de dados é o que chamamos também de dicionários, são estruturas onde temos um index

No caso de dicionarios o comum é usar uma string para servir de indexador:

```
const phones = {
  home: { country: "+1", area: "211", number: "652-4515" },
  work: { country: "+1", area: "670", number: "752-5856" },
  fax: { country: "+1", area: "322", number: "525-4357" },
}
```

Delarando com notação de tipos:

```
const phones: {
  [k: string]: {
    country: string
    area: string
    number: string
  }
} = {}
```

Arrays

No TS o jeito mais comum de descrever Arrays é com `[]`, mas tem o uso do `Array<T>`.

Implicitamente:

```
const fileExtensions = ["js", "ts"]
```

Explicitamente:

```
const cars = [
    {
        make: "Toyota",
        model:
            "Corolla",
        year:
            2002,
    }
]
const files = string[]
```

- Ou seja: `type[]` > `string[]`

Tuples

As tuplas são um tipo especial de lista, ela são ordenadas e cada item significa ou tem uma convenção diferentes:

> ⚠ A inferencia não trabalha bem com tuplas
>
> ```
> let myCar = [2002, "Toyota", "Corolla"]
> const [year, make, model] = myCar
> ```

Com Isso precisamos definir o estado do tipo da tupla na declaração:

```
let myCar: [number, string, string] = [
  2002,
  "Toyota",
  "Corolla",
]
// ERROR: not the right convention
myCar = ["Honda", 2017, "Accord"]
```

Para tornar-mos uma tupla apenas de leitura, o que é muito comum para as tuplas, deixar elas apenas para leituras

- Para isto usamos a keyword: `readonly`:

```
const roNumPair: readonly [number, number] = [4, 5];
roNumPair.length;

// Isso vai resultar em um erro, já que esse tipo de operação não existe
quando usamos readonly
roNumPair.push(6) // [4, 5, 6]
// Ao tentarmos excluir um item, vai retornar na mesma questão da linha
```

```
acima
roNumPair.pop() // [4, 5]
```

Functions and Returning Types

Para mexermos com funções e retornos é bom usarmos `type annotations`, onde definimos explicitamente os tipos, assim eles não recebem o tipo por inferencia, os parametros e o retorno das funções recebem o tipo padrão que é o `any`:

- Ao definirmos este código os parametros e o retorno da função tomam o tipo de `any`:

```
function add(a, b) {
  return a + b
}
```

- Agora está é uma boa prática e que faz sentido, ou seja, definirmos os tipo:
  - Com a type annotations: que é basicamente colocar : (dois pontos) e em seguida o tipo: `thing : type`;
  - Assim temos uma noção clara do que a função não só espera receber, mas o que ela retorna também

```
function add(a : number, b : number) : number {
  return a + b
}
```

Using the any Type

Tem que tomar cuidado para usar e o melhor é nem usar porque ela trás de volta a liberdade que tem no JS que o TS por motivo de existência vai contra

- Uso como type annotation:

```
function calculateTax(amount: any): any {
    return (amount * 1.2).toFixed(2);
}
let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;
console.log(`Price: ${price}`);
console.log(`Full amount in tax: ${taxAmount}`);
console.log(`Half share: ${halfShare}`);
```

O uso implicito também pode gerar problemas, como por exemplo quando não definimos nenhum tipo estatico para uma variavel ela receber por default any, caso só seja criada, ou retornos de funções ou parametros senão definidos os tipos podem ser consumidos como any:

```
function calculateTax(amount): any {
    return `$${(amount * 1.2).toFixed(2)}`;
}
let price = 100;
let taxAmount = calculateTax(price);
let halfShare = taxAmount / 2;
let personVal = calculateTax("Bob");
console.log(`Price: ${price}`);
console.log(`Full amount in tax: ${taxAmount}`);
console.log(`Half share: ${halfShare}`);
console.log(`Name: ${personVal}`)
```

Desabilitando o any implicito Basta usar essa compile option no `tsconfig.json`: `"noImplicitAny": true`

Using Type Union

Podemos unir o que se é igual formar seu proprio tipo:

> ⚠ Para fazer a union type usamos o caractere de bar ou pipe (|)

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}
let taxNumber = calculateTax(100, false);
let taxString = calculateTax(100, true)
```

A função `calculateTax` não retorna `number` e nem `string` e sim retorna `string|number`

Ou seja, só podemos usar os metodos que são presentes nos dois tipos, no caso, o código acima dara problema porque o método que eles compartilhar (`string` e `number`) na verdade é o `toString`

E podemos usar o `switch`:

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
```

```
        break;
}
```

Using Type Assertions

O type assertions é uma maneira de dizer ao compilador um tipo especifico para um valor

> Esse é um bom jeito de "burlar" o union type para que definamos que o valor será exatamente `number` por exemplo e não `string|number`, lembrando que o tipo a ser asertado deve ser um dos tipos da union

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}


let taxNumber = calculateTax(100, false) as number;
let taxString = calculateTax(100, true) as string;

console.log(`Number Value: ${taxNumber.toFixed(2)}`);
console.log(`String Value: ${taxString.charAt(0)}`);
```

> O `as` é a keyword para fazer a assertation e o `number` é o tipo alvo que queremos

> Temos uma sintaxe alternativa para a assertation, mas é não é usada quando mexemos com TSX por exemplo, por ela ter uma sintaxe como elemento do HTML:
>
> ```
> let taxString = <string> calculateTax(100, true);
> ```

Using type guard

Com verificar se o tipo especifico quando são tipos primitivos podemos usar a keyword `typeof` que retorna o tipo pertencente a aquele valor

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

if (typeof taxValue === "number") {
    console.log(`Number Value: ${taxValue.toFixed(2)}`);
} else if (typeof taxValue === "string") {
    console.log(`String Value: ${taxValue.charAt(0)}`);
}
```

⚠ O compilador não implementa o `typeof` isso vem do JS

Usando o Type Guard com switch

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
```

```
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
        break;
}
```

Using the Never Type

O TS prove o tipo `never` para situações em onde um type guard, tenha uma maneira de lidar com todos os tipos possiveis de tipos para um valor

Assim é possivel lidar com tipos inesperados que possam por algum mótivo, por exemplo se cair no `default` do switch do type guard:

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
        break;
    default:
        let value: never = taxValue;
        console.log(`Unexpected type for value: ${value}`);
}
```

Using the unknow type

TS também suporta um tipo desconhecido (unknown) que é uma maneira alternativa segura para o uso do `any`

Um valor desconhecido só pode ser atribuido a `any` ou a si proprio, a menos que seja usada uma asserção de tipo ou um type guard

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
        break;
    default:
        let value: never = taxValue;
        console.log(`Unexpected type for value: ${value}`);
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult;

console.log(`Number value: ${myNumber.toFixed(2)}`);
```

Um valor de tipo unknow não pode ser atribuido a outro tipo sem uso da type assertion:, senão produz esse erro:

```
src/index.ts(18,5): error TS2322: Type 'unknown' is not assignable to
type 'number'.
```

Assim para resolver isso, como foi falado, devemos usar a type assertion:

```
function calculateTax(amount: number, format: boolean): string | number
{
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue = calculateTax(100, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
        break;
    default:
        let value: never = taxValue;
        console.log(`Unexpected type for value: ${value}`);
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;

console.log(`Number value: ${myNumber.toFixed(2)}`);
```

Using Nullable Types

- null -> usado para representar algo que não existe ou é invalido e somente pode ser atribuido com o valor `null`
- undefined -> usado quando a variavel foi definida mas não foi atribuida nada nela ainda e só pode receber o valor `undefined`

```
function calculateTax(amount: number, format: boolean): string | number
{
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number = calculateTax(0, false);

switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
        break;
    default:
        let value: never = taxValue;
        console.log(`Unexpected type for value: ${value}`);
}

let newResult: unknown = calculateTax(200, false);
let myNumber: number = newResult as number;

console.log(`Number value: ${myNumber.toFixed(2)}`);
```

Restringindo atribuições nulas

Podemos definir uma compile option para tratar a questão de quando definimos uma atribuição nula, ou seja, quando não vai ser admitido a atribuição do tipo `null`, sendo essa a option:

> ⚠ "strictNullChecks": true

Ou seja, o código acima ao adicionar a option, daria este erro:

```
src/index.ts(3,9): error TS2322: Type 'null' is not assignable to type
'string | number'.
```

Para contornar isso podemos usar a type union anterior com o null, ou seja, temos isto:

```
function calculateTax(amount: number, format: boolean): string | number
| null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}
let taxValue: string | number | null = calculateTax(0, false);
switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
        break;
    default:
        if (taxValue === null) {
            console.log("Value is null");
        } else {
            console.log(typeof taxValue);
            let value: never = taxValue;
```

```
            console.log(`Unexpected type for value: ${value}`);
        }
}
```

Isso acaba com o erro que teriamos com a questão da restrição nula

Removendo o null da union com uma assertion

Para removermos o null que foi adiciona ao type union anterior podemos fazer o seguinte:

> ⚠ Uma assertion não nula deve ser usada somente quando sabemos que o valor `null` não pode ocorrer Um erro de tempo de execução pode ser causado se você aplicar a assertion e o valoor `null` pode ocorrer Uma maneira segura de fazer isso funcionar é um type guard

```
function calculateTax(amount: number, format: boolean): string | number
| null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number = calculateTax(100, false)!;

switch (typeof taxValue) {
    case "number":
        console.log(`Number Value: ${taxValue.toFixed(2)}`);
        break;
    case "string":
        console.log(`String Value: ${taxValue.charAt(0)}`);
        break;
    default:
        if (taxValue === null) {
```

```
            console.log("Value is null");
        } else {
            console.log(typeof taxValue);
            let value: never = taxValue;
            console.log(`Unexpected type for value: ${value}`);
        }
}
```

A assertion não nula é feita aqui:

> ⚠ `let taxValue: string | number = calculateTax(100, false)!;`
>
> Ela é feita com o sinal de exclamação (!)

Removendo null de uma union com type guard

Essa é uma maneira de lidar para erro em runtime (*tempo de execução*) por exemplo:

```
function calculateTax(amount: number, format: boolean): string | number
| null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue: string | number | null = calculateTax(100, false);

if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log(`Number Value: ${taxValue.toFixed(2)}`);
            break;
        case "string":
            console.log(`String Value: ${taxValue.charAt(0)}`);
            break;
```

```
    }
} else {
    console.log("Value is not a string or a number");
}
```

Using the definite assignment assertion

Se a option de `strictNullChecks` estiver ativada o compilador vai reportar um erro se a variavel é usada antes de atribuir valor

Isso é uma feature útil, mas pode existir tempos onde o valor é atribuido no caminho e isso não é visivel para o compilador:

```
function calculateTax(amount: number, format: boolean): string | number
| null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}
let taxValue: string | number | null;
eval("taxValue = calculateTax(100, false)");
if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log(`Number Value: ${taxValue.toFixed(2)}`);
            break;
        case "string":
            console.log(`String Value: ${taxValue.charAt(0)}`);
            break;
    }
} else {
    console.log("Value is not a string or a number");
}
```

A função `eval()` recebe uma string e executa esse conteúdo da string como statement do JavaScript (ele entende que é um código), mas o problema é que o compilador não enxerga que por exemplo o `taxValue recebe valor com o uso do eval()`

```
eval("taxValue = calculateTax(100, false)");
```

Assim o código acima dara erro:

```
src/index.ts(12,5): error TS2454: Variable 'taxValue' is used before
being assigned.
src/index.ts(13,9): error TS2322: Type 'string | number | null' is not
assignable to type
'string | number'.
  Type 'null' is not assignable to type 'string | number'.
src/index.ts(13,44): error TS2454: Variable 'taxValue' is used before
being assigned.
src/index.ts(14,20): error TS2454: Variable 'taxValue' is used before
being assigned.
```

A definição da assignment assertion diz para o TS que o valor será atribuido antes da variavel ser usada:

```
function calculateTax(amount: number, format: boolean): string | number
| null {
    if (amount === 0) {
        return null;
    }
    const calcAmount = amount * 1.2;
    return format ? `$${calcAmount.toFixed(2)}` : calcAmount;
}

let taxValue!: string | number | null;

eval("taxValue = calculateTax(100, false)");

if (taxValue !== null) {
    let nonNullTaxValue: string | number = taxValue;
    switch (typeof taxValue) {
        case "number":
            console.log(`Number Value: ${taxValue.toFixed(2)}`);
            break;
```

```
        case "string":
            console.log(`String Value: ${taxValue.charAt(0)}`);
            break;
    }
} else {
    console.log("Value is not a string or a number");
}
```

O definitive assignment assertion é o ponto de exclamação (!) e ele é usado depois do nome da variavel a ser usada

Assim, o código acima retornara com o sucesso o esperado:

```
Number Value: 120.00
```

Testing and Debugging TypeScript

Debuggingd de código TS

Primeiro devemos preparar o ambiente para debugging

Para isso adicionaremos uma compiler option chamada `sourceMap` como `true` para que o compilador passar compilando os arquivos TS e também fazendo um mapa dos arquivos, que tem a extensão de **map**, ao longo de todos os arquivos JS da pasta `dir`

- `tsconfig.jdon`

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true,
    "module": "commonjs",
    "sourceMap": true
  }
}
```

Adicionando breakpoints

Os breaktpoints é um simbolo que colocamos em determinada linha para dizer que ali será um ponto de parada para a plicação na hora que estiver debugando assim conseguimos saber o seu valor em tempo real, por exemplo

> ⚠ Usamos a keyword `debugger` ou o breakpoint de forma visual que é implementado na sua IDE

```
import {sum} from "./calc";

let printMessage = (msg: string): void => console.log(`Message:
```

```
${msg}`);
let message = ("Hello, TypeScript");

printMessage(message);

debugger;

let total = sum(100, 200, 300);
console.log(`Total: ${total}`);
```

Usando o Debugger integrado do NodeJS

É possivel usarmos o proprio debugger que vem junto ao NodeJS, temos que usar:

- **Iniciar depuração** Execute:

  ```
  node inspect dist/index.js
  ```

  Isso abre o depurador embutido do Node, carrega o arquivo compilado e pausa a execução antes da primeira linha. Você verá o prompt `debug>`.

- **Continuar execução** No prompt, digite:

  ```
  c
  ```

  (abreviação de *continue*). O programa segue até o próximo *breakpoint* ou instrução `debugger`.

- **Avaliar expressões** Use:

  ```
  exec("expressao")
  ```

  Exemplo:

  ```
  exec("message")
  ```

  Mostra o valor da variável `message`. As expressões precisam estar entre aspas.

- **Ajuda** Digite:

  ```
  help
  ```

  Lista os comandos disponíveis, como avançar (`n`), entrar em função (`s`), definir *breakpoints* (`sb`) e inspecionar a pilha de chamadas.

- **Sair** Pressione **Ctrl+C duas vezes**:

  - A primeira interrompe o comando atual.
  - A segunda encerra o depurador e volta ao prompt normal.

Using Linter

Linter é uma ferramenta para checar o código dos arquivo para verificar e validar se estão seguindo as regras que foram descritas para evitar problemas

- Para instalar o linter do TS:

> ⚠ npm install --save-dev tslint@5.16.0

Aqui está uma versão em português, simplificada e direta, sobre o uso do **TSLint** e suas regras:

Conjuntos de regras predefinidas

Nome	Descrição
tslint:recommended	Conjunto sugerido pela equipe do TSLint, indicado para uso geral em projetos TypeScript.
tslint:latest	Estende o recomendado, incluindo regras mais recentes.
tslint:all	Inclui todas as regras, podendo gerar muitos erros.

Executando o linter

```
npx tslint --project tsconfig.json --config tslint.json
```

- **--project**: usa o `tsconfig.json` para localizar os arquivos.
- **--config**: define o arquivo de configuração (`tslint.json`).

Exemplos de erros comuns

- **eofline**: exige que o arquivo termine com uma nova linha.
- **prefer-const**: recomenda usar `const` em vez de `let` quando o valor não muda.
- **no-console**: não permite chamadas a `console.log`.
- **no-debugger**: não permite uso da palavra-chave `debugger`.

Desabilitando regras no projeto

No arquivo `tslint.json`:

```
{
  "extends": ["tslint:recommended"],
  "linterOptions": {
    "format": "verbose"
  },
  "rules": {
    "eofline": false,
    "no-console": false,
    "prefer-const": false
  }
}
```

- `false` desativa a regra.
- É possível configurar regras específicas ou desligar todas.

Desabilitando regras em uma linha específica

```
// tslint:disable-next-line no-debugger
debugger;
```

- Aplica apenas à próxima instrução.
- Também existe `tslint:disable` para desativar regras em todo o arquivo.

Observações

- Linters ajudam a detectar erros reais e manter consistência.
- Podem gerar conflitos quando usados para impor estilo pessoal (indentação, chaves, espaços).
- Melhor prática: usar linting para problemas que afetam manutenção e deixar estilo para o editor/formatador automático.

Unit Tests

Alguns frameworks de testes unitários proveem suporte para TS, mas isso não é um morango

Isso quer dizer que, suportar TS para testes unitários significa permitir tests sejam definidos no TS e também automaticamente compilado o código TS antes de testar. Testes unitarios são feitos para executar pequenas partes da aplicação e isso pode ser feito somente com o JavaScript deste os ambiente de tempo de execução JavaScript não tem conhecimento das features do TS

Isso implica que as features TS não podem ser testadas

> ⚠ Vamos estar usando o Jest (https://jestjs.io/)

Adicionando o package do Jest ao projeto

```
npm install --save-dev jest@24.7.1
npm install --save-dev @types/jest
npm install --save-dev ts-jest@24.0.2
```

Comando	Função
npm install --save-dev jest@24.7.1	Instala o Jest (versão 24.7.1) como dependência de desenvolvimento. O Jest é o framework de testes.
npm install --save-dev @types/jest	Instala as definições de tipos do Jest para TypeScript, permitindo autocompletar e validação de tipos nos testes.
npm install --save-dev ts-jest@24.0.2	Instala o ts-jest, que é um preprocessor para permitir que o Jest entenda e execute arquivos .ts diretamente.

Configurando o framework de testes

Para configurar o Jest, adicionar um arquivo nomeado `jest.config.js` na pasta raiz do projeto com o conteúdo abaixo:

```
module.exports = {
    "roots": ["src"],
    "transform": {"^.+\\.tsx?$": "ts-jest"}
}
```

Métodos de asserção do Jest:

Método	Descrição
toBe(value)	Verifica se o resultado é igual ao valor especificado (não precisa ser o mesmo objeto).
toEqual(object)	Verifica se o resultado é o mesmo objeto/estrutura que o valor e specificado.
toMatch(regexp)	Verifica se o resultado corresponde à expressão regular fornecid a.
toBeDefined()	Verifica se o resultado está definido.
toBeUndefined()	Verifica se o resultado não está definido.
toBeNull()	Verifica se o resultado é `null`.
toBeTruthy()	Verifica se o resultado é avaliado como verdadeiro (*truthy*).
toBeFalsy()	Verifica se o resultado é avaliado como falso (*falsy*).
toContain(substri ng)	Verifica se o resultado contém a substring especificada.
toBeLessThan(val ue)	Verifica se o resultado é menor que o valor especificado.
toBeGreaterThan (value)	Verifica se o resultado é maior que o valor especificado.

Iniciando o framework

```
npx jest --watchAll
```

Using Functions

Redefinindo funções

No TS se queremos redefinir uma função algo como polimorfismo, onde temos o mesmo nome da função mas a sua assinatura é diferente. Ou seja, o número de parametros é diferente podemos simplesmente deixar o mesmo nome da função e mudarmos a quantidade de parametros:

```
function calculateTax(amount) {
    return amount * 1.2;
}

function calculateTax(amount, discount) {
    return calculateTax(amount) - discount;
}

let taxValue = calculateTax(100);
console.log(`Total Amount: ${taxValue}`);
```

⚠ O que estamos fazendo se chama, sobrecarga de função

O que disso que temos é que temos uma mesma função podendo ser executada de formas diferentes com base na

Using function parameters

```
function calculateTax(amount, discount) {
    return (amount * 1.2) - discount;
}

let taxValue = calculateTax(100, 0);

console.log(`2 args: ${taxValue}`);
taxValue = calculateTax(100);
```

```
console.log(`1 arg: ${taxValue}`);
taxValue = calculateTax(100, 10, 20);

console.log(`3 args: ${taxValue}`)
```

⚠ Teremos o compilador reportar esse problema ao tentar executar:

> ⚠ error TS2554: Expected 2 arguments, but got 1.
>
> error TS2554: Expected 2 arguments, but got 3.

```
function calculateTax(amount, discount) {
    return (amount * 1.2) - discount;
}

let taxValue = calculateTax(100, 0);

console.log(`2 args: ${taxValue}`);
taxValue = calculateTax(100);

console.log(`1 arg: ${taxValue}`);
taxValue = calculateTax(100, 10, 20);

console.log(`3 args: ${taxValue}`)
```

Optional parameters

Para termos parametros opcionais usamos o simbolo de interrogação (?):

⚠ function calculateTax(amount, discount?) {

Assim podemos criar um fallback caso ele não exista:

⚠ return (amount * 1.2) - (discount || 0);

```
function calculateTax(amount, discount?) {
    return (amount * 1.2) - (discount || 0);
}

let taxValue = calculateTax(100, 0);

console.log(`2 args: ${taxValue}`);
taxValue = calculateTax(100);
console.log(`1 arg: ${taxValue}`);

//taxValue = calculateTax(100, 10, 20);
//console.log(`3 args: ${taxValue}`)
```

ℹ É bom usar os parametros opcionais por último assim caso ele não venha evita a ordem que serem usados dar problema

Using parameter with a default value

Podemos definir valores default (padrão) para parametros assim temos um fallback já no código caso não seja passado

```
function calculateTax(amount, discount = 0) {
    return (amount * 1.2) - discount;
}

let taxValue = calculateTax(100, 0);
console.log(`2 args: ${taxValue}`);

taxValue = calculateTax(100);
console.log(`1 arg: ${taxValue}`);
//taxValue = calculateTax(100, 10, 20);
//console.log(`3 args: ${taxValue}`);
```

Assim temos que valores default tornam esse parametro optional, já que se não for definido ele usara o fallback

Using a rest parameter

A contra partida de parametros opcionais é o rest parameters, que permite uma função aceitar um número variavel de argumentos extras que são agrupados juntos e apresentados juntos

Uma função pode ter somente um rest parameter

```
function calculateTax(amount, discount = 0, ...extraFees) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(100, 0);
console.log(`2 args: ${taxValue}`);

taxValue = calculateTax(100);
console.log(`1 arg: ${taxValue}`);

taxValue = calculateTax(100, 10, 20);
console.log(`3 args: ${taxValue}`)
```

O rest parameters é criado com o três pontos (ellipsis) `...` antes do nome do parametro:

> ⚠ function calculateTax(amount, discount = 0, ...extraFees) {

- Os parametros seguem a ordem deles conforme a ordem
- Qualquer argumento adicional que não tenha parametro correspondente vai para o rest parameter
- O parametro sempre será um array
  - Senão tiver argumentos extras, será passado um array vazio `[]`

- Se houver, conterá todos os valores adicionais

Applying type annotations to function parameters

Por padrão o compiler define o type dos parameters como null, mas podemos deixar tudo mais específico

Para isso basta fazermos o normal e definirmos um type com dois pontos e o type na frente do nome dop parametro:

> ⚠ function calculateTax(amount: number, discount: number = 0, ...extraFees: number[])

Para um parametro defaul o type annotation vem antes:

> ⚠ discount: number = 0

E para um rest parameter, costuma ser apenas um array de algum type:

> ⚠ ...extraFees: number[]

```
function calculateTax(amount: number, discount: number = 0,
...extraFees: number[]) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(100, 0);
console.log(`2 args: ${taxValue}`);

taxValue = calculateTax(100);
console.log(`1 arg: ${taxValue}`);

taxValue = calculateTax(100, 10, 20)
console.log(`3 args: ${taxValue}`);
```

```
taxValue = calculateTax(100, 10, 20, 1, 30, 7);
console.log(`6 args: ${taxValue}`)
```

Controlling null parameters values

Para controlar para que não usar valores null ou undefined

```
function calculateTax(amount: number, discount: number = 0,
...extraFees: number[]) {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(null, 0);

console.log(`Tax value: ${taxValue}`);
```

Understanding Function Results

O compilador do TS para os retornos tenta inferir o tipo automaticamente, com base no código

- Se uma função retornar mais de um valor o compilador irá usar o type union para os tipos inferidos

> As declarações como essas dos tipos de retorno ficam dentro do arquivo de declaração ou os arquivos `.d.ts` e lá vai estar definido a type union da função por exemplo

Disabling implicit returns

Para desabilitarmos o retorno implicito e assim forçar para que todo retorno tenha seu type declarado devemos adicionar uma compiler option que vai fazer com que tenhamos isso:

⚠ "noImplicitReturns": true

- tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2018",
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true,
    "strictNullChecks": true,
    "noImplicitReturns": true
  }
}
```

Using typer annotations for function results

Isso funciona para explicitarmos o tipo de retorno:

⚠ function calculateTax(amount: number, discount: number = 0, ...extraFees: number[]): number

```
function calculateTax(amount: number, discount: number = 0,
        ...extraFees: number[]): number {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

let taxValue = calculateTax(100, 0);

console.log(`Tax value: ${taxValue}`)
```

Defining void functions

Funções que não retornam nada são procedimentos e procedimentos tem o seu tipo como `void`

```
function calculateTax(amount: number, discount: number = 0,
        ...extraFees: number[]): number {
    return (amount * 1.2) - discount
        + extraFees.reduce((total, val) => total + val, 0);
}

function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

writeValue("Tax value",  calculateTax(100, 0))
```

Overloading Function Types

O type union pode fazer iso possivel para definir um range de types para parametros de funções e retornos, mas eles não permitem o relacionamento entre eles para ser expressado de forma certa:

```
function calculateTax(amount: number | null): number | null {
    if (amount != null) {
        return amount * 1.2;
    }
    return null;
}

function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

let taxAmount: number | null = calculateTax(100);
if (typeof taxAmount === "number") {
    writeValue("Tax value",  taxAmount);
}
```

Assim podemos fazer a sobrecarga de tipos com o TS, basta definir a função e tipar o que queremos:

```
function calculateTax(amount: number): number;
function calculateTax(amount: null): null;
function calculateTax(amount: number | null): number | null {
    if (amount != null) {
        return amount * 1.2;
    }
    return null;
}
```

```
function calculateTax(amount: number): number;
function calculateTax(amount: null): null;
function calculateTax(amount: number | null): number | null {
    if (amount != null) {
        return amount * 1.2;
    }
    return null;
}

function writeValue(label: string, value: number): void {
    console.log(`${label}: ${value}`);
}

let taxAmount: number = calculateTax(100);
//if (typeof taxAmount === "number") {
    writeValue("Tax value",  taxAmount);
//}
```

Using Arrays

Working with Arrays

Para trabalharmos com arrays é simples, basta especificarmos o array type depois do nome:

- Definimos o array type `number[]` com o nome do tipo e colchetes []

```
let prices: number[] = [100, 75, 42];
let names: string[] = ["Hat", "Gloves", "Umbrella"];
```

> Podemos usar a notação de array para conter múltiplos tipos como a union type por exemplo e basta usarmos parenteses `()`
>
> ```
> let array : (number | string)[];
> ```

Performing Operations on Typed Arrays

```
function calculateTax(amount: number): number {
    return amount * 1.2;
}

function writePrice(product: string, price: number): void {
    console.log(`Price for ${product}: $${price.toFixed(2)}`);
}

let prices: number[] = [100, 75, 42];
let names: string[] = ["Hat", "Gloves", "Umbrella"];

prices.forEach((price: number, index: number) => {
```

```
    writePrice(names[index], calculateTax(price));
})
```

Temos uma outra sinxtas para aarrays

Está sintaxe pode ser escrita da seguinte forma:

```
let prices : Array<number> = [100, 45, 560];
```

Que é o mesmo que escrevermos assim:

```
let prices : number[] = [100, 45, 560];
```

O problema reside em não podermos usar essa sintaxe em arquivos TSX, justamente por causa que ele pode confundir os brackets como tag (`Array<number>`). Assim é melhor optar pela sintaxe do square bracket (colchetes)