

Objetivo:

- I. Node.js;
- II. NPM – Node Package Manager;
- III. Linguagem TypeScript;
- IV. Estrutura de um projeto Node.js;
- V. Projeto TypeScript no VS Code;
- VI. Compilar e executar arquivo TypeScript;
- VII. Arquivo tsconfig.json;
- VIII. ECMAScript;
- IX. Declaração de variável no TypeScript;
- X. Tipos de dados do TypeScript;
- XI. União de tipos no TypeScript;
- XII. Função no TypeScript;
- XIII. Export e import.

I. Node.js

O Node.js é uma plataforma de execução de código que permite executar código JS (JavaScript) fora do navegador. O Node.js utiliza o mecanismo de JS do Google Chrome, chamado V8, para executar o código JS diretamente no servidor ou em qualquer outro dispositivo que tenha o Node.js instalado, ou seja, para executarmos um programa JS no nosso computador precisamos ter instalado o Node.js.

Principais características do Node.js:

- Arquitetura orientada a eventos: o Node.js utiliza um modelo orientado a eventos e uma arquitetura de loop de eventos para lidar com solicitações assíncronas de maneira eficiente. Isso permite que o servidor (back-end) responda a várias solicitações simultaneamente, sem bloquear o processo;
- Input/Output (entrada/saída) não bloqueante: o Node.js usa chamadas de I/O não bloqueantes, o que significa que, em vez de esperar pela conclusão de uma operação de entrada/saída, ele passa para a próxima operação enquanto aguarda o resultado da anterior. Isso ajuda a manter o desempenho do aplicativo mesmo quando ocorrem operações lentas de I/O;
- NPM (Node Package Manager): o Node.js vem com o NPM, um gerenciador de pacotes poderoso que permite instalar, gerenciar e compartilhar bibliotecas de terceiros de forma fácil. O NPM é uma parte essencial do ecossistema do Node.js e oferece acesso a milhares de módulos e bibliotecas prontas para uso;
- Construção de APIs e aplicativos de servidor: o Node.js é frequentemente usado para criar APIs RESTful e aplicativos de servidor. Com sua abordagem assíncrona e capacidade de lidar com várias conexões simultâneas, ele é uma escolha popular para aplicativos que exigem alto desempenho e escalabilidade;
- Ecossistema rico: o Node.js tem um ecossistema muito ativo e uma grande comunidade de desenvolvedores. O repositório da plataforma NPM possui uma vasta gama de bibliotecas JS.

Ao contrário do JS tradicional que é executado no navegador, para fazermos uso do JS fora do navegador precisaremos instalar o Node.js no nosso computador. Acesse <https://nodejs.org/en/download> e faça o download do instalador na versão **.msi** para Windows. Após instalar, será criada a pasta nodejs (Figura 1) e na pasta C:\Program Files\nodejs nodejs\node_modules\npm estará a instalação do gerenciador de pacotes da linguagem JS (npm – Node Package Manager), isto é, o npm será instalado juntamente com o Node.js.

Forneça os comandos da Figura 2 no CMD (prompt do DOS) para verificar as versões e, por consequência, verificar se as instalações foram efetivadas.

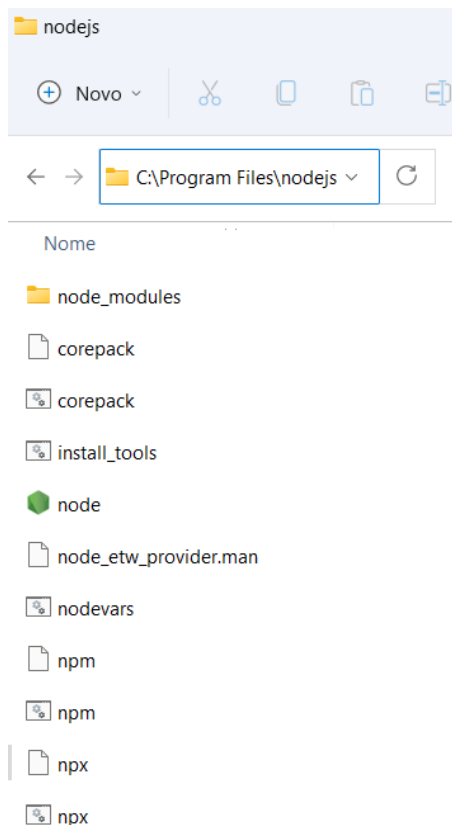


Figura 1 – Arquivos da pasta Node.js no computador. Deverá estar na pasta C:\Program Files\nodejs.

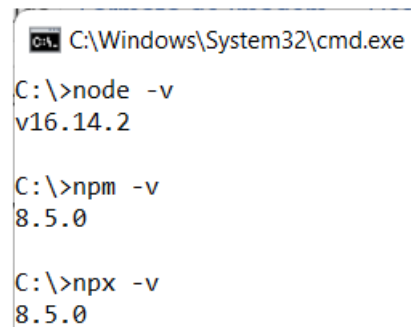


Figura 2 – Comandos para obter a versão do Node.js, npm e npx. Em todos os casos, executamos programas por linha de comando (CLI – Command Line Interface).

II. NPM – Node Package Manager

npm é a maior plataforma para o registro de software do mundo. Desenvolvedores de código aberto usam o npm para compartilhar pacotes, e muitas organizações também usam o npm para gerenciar o desenvolvimento privado.

O npm consiste em três partes distintas:

1. Website: portal com a documentação dos pacotes. Como exemplo, <https://www.npmjs.com/package/express> possui a documentação do pacote fornecida pelo responsável pelo desenvolvimento do pacote Express;
2. Command Line Interface (CLI): ferramenta para fazer o download, instalar e gerenciar os pacotes e módulos na sua aplicação. Ele coloca os módulos no local que o node pode encontrá-los e gerencia os conflitos de dependência (na prática, um pacote pode depender de vários outros, desta forma, ao instalar um pacote o npm faz o download e instalação dos pacotes dependentes). O npm é uma ferramenta de linha de comando (CLI), ou seja, o programa é executado através de comandos digitados no prompt, terminal ou qualquer outro ambiente de linha de comando.

O comando npm a seguir baixa o pacote do repositório da npm e instala na pasta **node_modules** do projeto, ou seja, estará disponível apenas para o projeto:

```
npm install packagename (faz o download e instala o pacote)
npm install packagename1 packagename2 (é possível instalar mais de um pacote)
npm i packagename (forma abreviada)
```

O comando npm a seguir instala o pacote nas dependências de desenvolvimento do projeto. O pacote estará na pasta node_modules do projeto, mas não fará parte da versão de produção a ser entregue para o cliente. Os pacotes de desenvolvimento são aqueles que auxiliam apenas o programador na codificação e não serão necessários na execução do programa a ser entregue para o cliente:

```
npm i packagename -D
```

O comando npm a seguir instala o pacote globalmente no computador, ou seja, na pasta C:\Program Files\nodejs\node_modules e estará disponível para qualquer projeto no computador:

```
npm i packagename -g
```

O npm CLI é formado por vários comandos, sendo que o comando npx (eXecute - npm package runner) é uma ferramenta para executar pacotes Node. O npx será necessário para criarmos aplicações, assim como o React. A seguir tem-se o comando para executar um pacote usando npx:

```
npx packagename
```

3. Registro: repositório de softwares JS e TS (pacotes e módulos) e metadados.

Em outras palavras, o website funciona como um portal de divulgação dos softwares para os usuários; o programa por linha de comando é usado pelos programadores, pois o CLI é usado para baixar e instalar as bibliotecas de códigos; e o registro é o repositório onde as bibliotecas ficam armazenadas.

Para mais detalhes <https://docs.npmjs.com/about-npm>.

III. Linguagem TypeScript

JavaScript (JS) e TypeScript (TS) são duas linguagens de programação relacionadas, mas com algumas diferenças importantes. Aqui estão as principais diferenças entre JS e TS:

- **Tipagem estática:** JS é uma linguagem de programação de tipagem dinâmica, o que significa que as variáveis não têm um tipo definido e podem ser alteradas durante a execução do programa. Já o TS é uma linguagem de programação de tipagem estática, o que significa que as variáveis têm um tipo definido em tempo de compilação (mais precisamente ao criar a variável) e não podem ser alteradas posteriormente. Tome como exemplo os códigos a seguir, no JS a variável **valor** tem o tipo de dado alterado em cada atribuição. No caso do TS, a variável **nro** tem o tipo de dado **number** definido na declaração e não poderá receber conteúdos de outros tipos de dados:

Código .js	Código .ts
<pre>// a variável é definida sem um tipo de dado definido // undefined é o tipo de dado da variável let valor; // number é o tipo de dado da variável valor = 12; // string é o tipo de dado da variável valor = "12";</pre>	<pre>// number é o tipo de dado da variável let nro:number; nro = 12; // a atribuição dará o seguinte erro: // o tipo 'string' não é atribuível ao // tipo 'number' nro = "12";</pre>

- O TS adiciona a capacidade de definir tipos para variáveis, parâmetros de função, retornos de função e outros elementos do código, o que pode ajudar a detectar erros em tempo de desenvolvimento. No exemplo a seguir a função `dif` só aceita parâmetros do tipo `number`. O código TS mostrará a mensagem de erro ao digitar, ou seja, não será necessário executar para obter o erro ao fazer a atribuição usando o tipo `string`:

Código .js	Código .ts
<pre>// as variáveis a e b recebem valores // de qualquer tipo function somar(a,b){ return a + b; } // estamos chamando a função somar passando // dois parâmetros do tipo number let x = somar(2,3); // estamos chamando a função somar passando // dois parâmetros do tipo string let y = somar("o","i");</pre>	<pre>// as variáveis a e b podem receber // somente valores do tipo number function dif(a:number,b:number){ return a + b; } let r:number = dif(2,3); // os valores passados como parâmetros // dará erro por serem string let s:number = dif("1","2");</pre>

- Compilação: JS é uma linguagem interpretada, o que significa que o código é executado diretamente por um interpretador JS. Por outro lado, o TS é uma linguagem compilada. Antes de ser executado, o código TS precisa ser compilado para JS. A compilação é realizada através do compilador `tsc` (TypeScript Compiler), que verifica a sintaxe, a tipagem e gera o código JS equivalente. Em outras palavras, não existe código TS em tempo de execução, ele precisa ser transformado para código JS para ser executado. No exemplo a seguir, criou-se o arquivo `index.ts`, na sequência usou-se o comando `tsc ./index.ts` para gerar o arquivo `index.js`. Observe que o arquivo `.js` não possui os recursos de tipos de dados:

EXPLORER

EXEMPLO

- TS index.ts
- package.json

TS index.ts

```
1 function dif(a:number,b:number){
2     return a + b;
3 }
4
5 let res:number = dif(2,3);
6
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

PS C:\Desktop\exemplo> tsc ./index.ts

Após executar o comando `tsc ./index.ts` será criado o arquivo `index.js` com o código a seguir:

```
function dif(a, b) {
    return a + b;
}
var res = dif(2, 3);
```

Observação: para testar esse exemplo você precisará executar os seguintes passos:

- Crie a pasta `exemplo` (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
- Abra a pasta no VS Code e acesse o terminal do VS Code;

3. No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node (arquivo `package.json`):

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> npm init -y
```

4. Certifique-se que você tenha instalado o typescript globalmente no seu computador digitando o comando `tsc --version`. Será exibido o número da versão instalada caso tenha instalado o typescript:

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> tsc --version
Version 4.9.5
```

5. Se não tiver instalado: execute o comando a seguir para instalar globalmente e após instalar execute o comando `tsc --version` para verificar a versão:

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
PS C:\Desktop\exemplo> npm i typescript -g
```

6. Digite o comando `tsc ./index.ts` para compilar o arquivo `index.ts` e gerar a versão JS.

- Recursos avançados: TS adiciona recursos avançados à linguagem JS. Ele oferece suporte a recursos como classes, interfaces, herança, módulos e genéricos (recursos abordados ao longo desta disciplina). Esses recursos permitem que os desenvolvedores apliquem conceitos de POO (Programação Orientada a Objetos) de forma mais estruturada e robusta;
- Ferramentas de desenvolvimento: TS possui um sistema de tipos que pode ser verificado durante o desenvolvimento por meio de ferramentas de lint (veja a seguir) como aquelas integradas no editor de código do VS Code. Essa verificação de tipos ajuda a identificar erros comuns e melhora a produtividade do desenvolvedor, fornecendo sugestões de autocompletar e validação de código mais robusta. No JS, a falta de tipagem estática pode tornar mais difícil a detecção de erros antes da execução do código. No exemplo a seguir, o VS Code alerta que o tipo de dado da variável `res` está incorreto:

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

Type 'number' is not assignable to type 'string'. ts(2322)

```
let res: string
```

[View Problem \(Alt+F8\)](#) No quick fixes available

```
let res:string = dif(2,3);
```

Lint é um termo utilizado na área de desenvolvimento de software para se referir a ferramentas que analisam o código-fonte em busca de erros, inconsistências, más práticas e problemas de estilo. Essas ferramentas são chamadas de "linters" ou "ferramentas de linting". No VS Code usaremos, posteriormente, a extensão ESLint

TS é um superset do JS. Em outras palavras, TS é JS + recursos de tipagem de dados

IV. Estrutura de um projeto Node.js

Um projeto Node.js é uma aplicação ou software desenvolvido usando a plataforma Node.js, que permite a execução de código JS do lado do servidor. Aqui estão alguns elementos comuns encontrados em um projeto Node.js:

- Estrutura de pastas: um projeto Node.js geralmente tem uma estrutura de pastas organizada com alguns diretórios, sendo eles:
 - src: contém os arquivos-fonte da aplicação, tais como, arquivos JS e TS. Esses arquivos precisam ser compilados (arquivos TS) e/ou interpretados (arquivos JS) no servidor, ou seja, não são arquivos enviados para o cliente da forma como eles se encontram;
 - public: contém arquivos estáticos que podem ser acessados diretamente pelo navegador, ou seja, esses arquivos não serão compilados ou executados no servidor. Geralmente essa pasta recebe arquivos de imagens, CSS, HTML e arquivos JS (enviados para o cliente da forma como eles foram digitados);
 - tests: contém arquivos de testes de código;
 - assets: não é uma pasta padrão de um projeto Node.js. Geralmente, usamos ela para manter arquivos de imagens, fontes, PDF etc. usados pelo código JS/TS da pasta src;
 - node_modules: contém as dependências instaladas pelo gerenciador de pacotes npm. Essa pasta é gerenciada pelo npm CLI e não deve ser modificada pelo programador.
- Arquivo package.json: é o arquivo de configuração fundamental do projeto Node.js. Ele descreve as informações do projeto, as dependências do pacote e contém comandos para a construção, execução e gerenciamento do projeto. Por ser um arquivo JSON (JavaScript Object Notation) os dados são organizados em propriedades. A seguir tem-se algumas propriedades que podem ser encontradas em um arquivo package.json:
 - name: especifica o nome do projeto;
 - version: indica a versão atual do projeto;
 - main: indica o arquivo JS/TS que deve ser carregado e executado quando o seu pacote é importado ou utilizado como uma dependência em outro projeto. Quando um pacote Node.js é importado ou requerido por outro código, o Node.js procura pelo arquivo especificado na propriedade main. Por exemplo, se a propriedade main for definida como index.js, o Node.js procurará pelo arquivo index.js no diretório raiz do pacote. Esse arquivo será o ponto de entrada do código do pacote;
 - dependencies: lista de pacotes e módulos externos necessários para o funcionamento do projeto durante o desenvolvimento e em ambiente de produção. As dependências podem ser instaladas e removidas usando a ferramenta npm;
 - devDependencies: lista de pacotes e módulos externos necessários apenas durante o desenvolvimento, como ferramentas de testes, bibliotecas de construção etc. Em outras palavras, esses pacotes e módulos não serão implantados na versão da aplicação disponibilizada para o ambiente de produção. As dependências de desenvolvimento podem ser instaladas e removidas usando a ferramenta npm;

- `scripts`: lista de comandos personalizados para automatizar tarefas comuns. Como exemplo, você pode definir um script "start" para iniciar o aplicativo, ou um script "dev" para iniciar o código a cada mudança;
- `author`: identifica o autor do projeto;
- `license`: especifica a licença sob a qual o projeto está disponível.
- Arquivo `package-lock.json`: é gerado automaticamente quando instalamos pacotes no projeto usando o npm. Esse arquivo possui o caminho exato de todos os pacotes e suas dependências transitivas no projeto, ou seja, um pacote pode depender de outros pacotes e será necessário instalar os pacotes dependentes. Desta forma, se excluirmos a pasta `node_modules` ou subirmos o projeto no GitHub sem a pasta `node_modules` e quisermos refazer o projeto, o comando `npm install` ou, sua forma condensada `npm i`, usará o arquivo `package-lock.json` para baixar e instalar todas as dependências nas versões especificadas.

V. Projeto TypeScript no VS Code

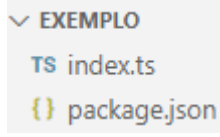
Siga os passos para criar um projeto Node.js usando TS:

1. Crie a pasta `exemplo` (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
2. Abra a pasta no VS Code e acesse o terminal do VS Code;
3. No terminal, execute o comando `npm init -y` para criar o arquivo fundamental de um projeto Node (arquivo `package.json`):

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> npm init -y
```

4. Crie o arquivo `index.ts` e digite o código mostrado a seguir:

Estrutura do projeto	Código do arquivo <code>index.ts</code>
	<pre>function dif(a:number,b:number){ return a + b; } let res:number = dif(2,3); console.log("Resultado", res);</pre>

VI. Compilar e executar arquivo TypeScript

O TS é um superset do JS, isso significa que é necessário compilar para JS antes de executar. Para isso, podemos usar o comando `tsc` (TypeScript Compiler) para compilar o arquivo TS em JS. Em seguida, podemos usar o interpretador de JS, como o Node.js, para executar o arquivo JS resultante.

Existem algumas formas para compilarmos e executarmos arquivos TS:

- a) Compilar e executar manualmente: o comando `tsc` (TypeScript Compiler) pode ser usado para compilar o arquivo TS e gerar a versão JS. Em seguida, podemos usar o interpretador `node` para executar o arquivo JS. A seguir tem-se os passos:
 - Certifique-se que você tenha instalado o typescript globalmente no seu computador digitando o comando `tsc --version`. Será exibido o número da versão instalada caso tenha instalado o typescript:

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> tsc --version
Version 4.9.5
```

- Se não tiver instalado: execute o comando a seguir para instalar globalmente e após instalar execute o comando `tsc --version` para verificar a versão:

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
PS C:\Desktop\exemplo> npm i typescript -g
```

- O comando `tsc` é um compilador. Use `tsc ./index.ts` para compilar o arquivo TS e gerar a versão JS;
- O comando `node` é um interpretador. Use `node ./index.js` para executar o arquivo resultante da compilação:

EXEMPLO

JS index.js

TS index.ts

{}

 package.json

```
TS index.ts > ...
1  function dif(a:number,b:number){
2      |   return a + b;
3      |
4      }
5
6  let res:number = dif(2,3);
   console.log("Resultado", res);
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
● PS C:\Desktop\exemplo> tsc ./index.ts ← Cria o arquivo index.js
● PS C:\Desktop\exemplo> node ./index.js ← Executa o arquivo index.js
Resultado 5
```

- b) Compilar e executar utilizando ferramentas de desenvolvimento: a ferramenta `ts-node` (<https://www.npmjs.com/package/ts-node>) evita a etapa de compilação manual. O `ts-node` é um interpretador TS que permite executar arquivos TS diretamente, sem a necessidade de compilar previamente. A seguir tem-se os passos:

- Use o comando a seguir para instalar o `ts-node` como dependência de desenvolvimento. Para executar a ferramenta `ts-node` precisaremos do pacote `typescript`, então instale ambos como dependência de desenvolvimento. Lembre-se que as dependências de desenvolvimento (propriedade `devDependencies`) são utilizadas somente durante a codificação:

Use os comandos a seguir para instalar os pacotes `ts-node` e `typescript` como dependências de desenvolvimento:

Conteúdo do arquivo `package.json` após instalar os pacotes:

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
PS C:\Desktop\exemplo> npm i -D ts-node
added 19 packages, and audited 20 packages in 6s
found 0 vulnerabilities

PS C:\Desktop\exemplo> npm i -D typescript
up to date, audited 20 packages in 607ms
found 0 vulnerabilities
```

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


Estrutura do projeto. O comando npm criou a pasta node_modules com o código das bibliotecas e o arquivo package-lock.json com o caminho e versão dos pacotes

EXEMPLO

```
> node_modules
TS index.ts
{} package-lock.json
{} package.json
```

```
"keywords": [],
"author": "",
"license": "ISC",
"devDependencies": {
  "ts-node": "^10.9.1",
  "typescript": "^5.1.3"
}
```

Observação: no nosso computador já temos o pacote typescript globalmente, então não temos a obrigação de instalar no nosso projeto. Porém, constitui boa prática instalar o pacote typescript como dependência do nosso projeto, pois poderemos enviar o projeto para ser executado numa máquina (dispositivo) que não tenha o typescript global.

- O **ts-node** compilará o arquivo **index.ts** em tempo de execução e o executará diretamente sem criar a versão JS:

```
TERMINAL  PROBLEMS  OUTPUT
PS C:\Desktop\exemplo> npx ts-node ./index
Resultado 5
PS C:\Desktop\exemplo> ts-node ./index
Resultado 5
```

Observação: o comando npx (**n**pm **e**xecute - npm package runner) é usado para executar pacotes e ferramentas Node.js. Nesse exemplo usamos o npx para executar o programa ts-node.

- Podemos colocar o comando **ts-node** na propriedade **scripts** do package.json do projeto. No exemplo a seguir chamamos a propriedade de **start**, porém poderia ser qualquer outro nome. Observe que o conteúdo da propriedade **start** é exatamente o comando que digitamos também pelo terminal:

Conteúdo do arquivo package.json após adicionar a propriedade **start** no **scripts**:

```
{
  "name": "exemplo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "ts-node ./index"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^10.9.1",
    "typescript": "^5.1.3"
  }
}
```

Use um dos comandos a seguir para executar o projeto através dos scripts de automação:

```
TERMINAL  PROBLEMS  OUTPUT
PS C:\Desktop\exemplo> npm start
> exemplo@1.0.0 start ← exemplo é o conteúdo da
> ts-node ./index      propriedade name e 1.0.0 é a
Resultado 5             propriedade version do arquivo
                        package.json
PS C:\Desktop\exemplo> npm run start
> exemplo@1.0.0 start
> ts-node ./index ← O comando npm start executa o
Resultado 5           comando que está na propriedade start
```

VII. Arquivo tsconfig.json

É um arquivo de configuração usado em projetos TS para definir as opções e configurações do compilador TS. Ele permite personalizar o processo de compilação e ajustar o comportamento do compilador para atender às necessidades do seu projeto.

Utilize o comando `tsc --init` para criar o arquivo `tsconfig.json` na raiz do projeto. O arquivo terá várias propriedades, mas somente alguns não estarão comentadas.

No próprio arquivo possui a descrição das propriedades.

TERMINAL PROBLEMS OUTPUT

```
PS C:\Desktop\exemplo> tsc --init
Created a new tsconfig.json with:
  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true
```

Apesar do arquivo ter várias propriedades, essas não estão comentadas

Observação: no exemplo atual não notaremos diferença na compilação usando ou não o `tsconfig.json`. Porém, constitui boa prática ter o arquivo `tsconfig.json` em um projeto TS.

VIII. ECMAScript

A ECMAScript (também conhecida como ES) é uma especificação de linguagem de programação que define a sintaxe e o comportamento das linguagens de script, como JS.

A ECMAScript passou por várias versões ao longo dos anos. Cada versão traz novos recursos para a linguagem JS, como classes, arrow functions, destructuring assignment, async/await, módulos, entre outros.

Até a 5ª versão, as ECMAScripts eram nomeadas por números es3, es4 e es5. Na 6ª versão admite-se o número es6 ou o ano de criação es2015. Porém, desde 2016, as versões são nomeadas por anos: es2016, es2017, es2018, ...

O Node.js é construído sobre o motor JavaScript V8, desenvolvido pelo Google, que é compatível com as especificações da ECMAScript. Ao atualizar o Node.js para versões mais recentes, será obtido suporte para as funcionalidades mais recentes do ECMAScript.

No contexto da compilação de TS para JS, a ECMAScript está relacionada à versão da ECMAScript para a qual o código TS será compilado. O TS é um superset da ECMAScript, o que significa que ele adiciona recursos adicionais ao JS, como tipos estáticos, interfaces, anotações de tipo, entre outros.

O arquivo de configuração `tsconfig.json` é usado para especificar as opções de compilação, incluindo a versão da ECMAScript para a qual o TS deve ser compilado. Como exemplo, ao definir `"target": "es2016"`, no arquivo `tsconfig.json`, o código TS será compilado para JS compatível com a 7ª versão da ECMAScript.

Para detalhes sobre as versões https://www.w3schools.com/js/js_versions.asp.

IX. Declaração de variável no TypeScript

Uma variável é um espaço na memória do computador. Uma variável é formada por:

- nome: identificador do espaço de memória;
- conteúdo: valor guardado no espaço de memória;
- tipo de dado: tipo de dado do valor guardado no espaço de memória.

Toda variável possui um tipo de dado, porém, este tipo pode ser definido de acordo com o conteúdo da variável (esse recurso recebe o nome de **tipagem dinâmica**). Como exemplo, a linguagem JS possui tipagem dinâmica:

```
let valor = 12; // o tipo de dado da variável é number
```

```
valor = "12"; // o tipo de dado da variável é string
```

Outra forma, é o tipo de dado ser definido na declaração (**tipagem estática**). Como exemplo, o TS possui tipagem estática. A vantagem da tipagem estática é o erro de atribuição ser detectado em tempo de compilação, ou seja, não precisamos executar o código para detectar o erro. No exemplo a seguir, a mensagem de erro foi exibida pelo fato do VS Code usar alguma ferramenta de lint (análise estática) para fazer a compilação ao digitarmos:

```
let valor:number = 12;
Type 'string' is not assignable to type 'number'. ts(2322)
let valor: number
View Problem (Alt+F8) No quick fixes available
valor = "12";
```

Na linguagem TS podemos definir o tipo de dados de uma variável de várias maneiras. A seguir tem-se as duas mais comuns:

1. Inferência de tipo: o TS pode inferir o tipo de uma variável com base no valor atribuído a ela. Sempre que possível o TS tenta automaticamente inferir o tipo de dado. Por exemplo:

```
let nome = "Ana"; // TS infere o tipo string
let idade = 25; // TS infere o tipo number
```

2. Anotação de tipo explícita (type annotations): o tipo é explicitamente anotado usando a sintaxe de dois pontos após o nome da variável. Por exemplo:

```
let nome:string = "Ana";
let idade:number = 25;
```

X. Tipos de dados do TypeScript

O TS possui os seguintes tipos de dados primitivos:

- string: sequências de caracteres;
- number: números inteiros ou reais;
- boolean: somente os valores true e false;
- null: representa um valor nulo. Na prática é usado para referenciar a ausência de endereço de memória. Uma analogia seria “qual é o endereço da sua casa de praia?” e você responde null, ou seja, você disse que não possui casa de praia;
- undefined: representa um valor indefinido (não definido);
- bigint: representa números inteiros maiores do que $-(2^{53} - 1)$ e menores do que $(2^{53} - 1)$.

Exemplos:

```
let nome:string = "Ana";
let idade:number = 25; //número inteiro
let peso:number = 59.9; //número real
let doador:boolean = true; //booleano
let fone:null = null; //endereço de memória
let cel:undefined = undefined; //não definido
let distancia:bigint = 20n; //o literal n é usado para indicar que o número é bigint
```

Observação: o literal bigint está disponível a partir da ECMAScript 2020. Desta forma, será necessário alterar a propriedade target do arquivo tsconfig.json para "target": "ES2020".

No TS todas as variáveis precisam ser declaradas com um tipo de dado, porém nem sempre sabemos os tipos de dados que a variável receberá durante a execução do código. Para contornar este problema, o TS possui o tipo especial **any**.

O tipo **any** é usado para indicar uma variável que pode ter qualquer tipo de valor. É uma forma de desabilitar a verificação de tipo estática fornecida pelo TS. Quando uma variável é declarada como tipo any, ela pode receber qualquer valor e realizar qualquer tipo de operação sem gerar erros de tipo durante a compilação.

No exemplo a seguir a variável `obj` pode receber qualquer tipo de valor:

```
let obj:any = 1;
console.log(typeof obj); // imprime number
obj = "oi";
console.log(typeof obj); // imprime string
obj = true;
console.log(typeof obj); // imprime boolean
```

Observações:

- O operador `typeof` retorna uma string com o nome do tipo de dado do operando;
- Recomenda-se evitar o uso excessivo do tipo **any**, pois a perda de verificação de tipo pode comprometer a segurança e a robustez do seu código.

Existem duas maneiras de declarar um tipo array no TS:

- Anotação de tipo com colchetes []:

```
let nros:number[] = [1, 2, 3];
let nomes:string[] = ["Pedro", "Maria", "Ana"];
let outros:any[] = [2.5, "texto", true];
```

- Usando a generics (tipos genéricos) `Array<T>`, onde `Array` é um tipo de dado objeto para arrays e `T` é o tipo de dado dos elementos do array:

```
let nros:Array<number> = [1, 2, 3];
let nomes:Array<string> = ["Pedro", "Maria", "Ana"];
let outros:Array<any> = [2.5, "texto", true];
```

Para mais detalhes <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html>.

XI. União de tipos no TypeScript

A união de tipos no TS é uma forma de combinar dois ou mais tipos em um único tipo, permitindo que uma variável possa aceitar diferentes tipos de valores.

A sintaxe para criar uma união de tipos é usando o operador de pipe (`|`) entre os tipos a serem unidos. No exemplo a seguir a variável pode receber valores dos tipos `number` e `string`:

```
let valor:number|string = 12;
console.log(typeof valor); // imprime number
```

```
valor = "oi";  
console.log(typeof valor); // imprime string
```

A união de tipos pode ser usada para os elementos do array. No exemplo a seguir os elementos dos arrays podem ser number ou string:

```
// recebeu um array onde os elementos são number e string  
let nros:(number|string)[] = [1, "oi", 3];  
let valores:Array<number|string> = [1, "oi", 3];
```

No exemplo anterior os tipos `(number | string)[]` foram agrupados usando parênteses para indicar que os elementos do array podem ser number ou string, mas se usarmos a notação a seguir, a variável `teste` poderá receber array de number[] ou array de string[]:

```
let teste:number[]|string[];  
teste = [1,2,3]; // recebeu um array de number  
teste = ["a", "b", "c"]; // recebeu um array de string
```

XII. Função no TypeScript

No TS podemos fazer a anotação de tipo de dado dos parâmetros e do retorno da função. A anotação de tipo de parâmetro e retorno não é obrigatório no TS, porém ela torna mais eficaz a checagem pelas ferramentas de análise estática em tempo de digitação, assim como as mensagens de erro exibidas pelo VS Code ao digitarmos o nosso programa.

Uma ferramenta de análise estática examina o código-fonte sem a necessidade de executá-lo. Elas são projetadas para identificar potenciais problemas, erros de sintaxe, inconsistências e padrões de código indesejados antes mesmo da execução do código.

Essas ferramentas realizam análises detalhadas do código em busca de problemas comuns, como erros de digitação, uso incorreto de variáveis, problemas de escopo, ausência de retornos em funções, entre outros. Elas também podem verificar se o código está em conformidade com as melhores práticas, convenções de estilo e diretrizes recomendadas.

Essas ferramentas ajudam a melhorar a qualidade e a legibilidade do código, identificando problemas em estágios iniciais do desenvolvimento e nos permitindo tomar medidas corretivas antes que os erros sejam encontrados em tempo de execução. Isso resulta em um código mais robusto, com menos erros e mais fácil de manter.

O ESLint é um plugin de análise estática para o VS Code.

No exemplo a seguir os parâmetros `a` e `b` foram anotados com o tipo `number`, assim como fazemos na declaração de variáveis. Já o tipo de retorno é colocado imediatamente após o par de parênteses:

```
// a função foi anotada com retorno void pelo fato de a função não ter return no seu corpo  
function somar(a:number,b:number):void {  
    const r = a + b;  
    console.log("Resultado", r);  
}
```

```
}  
  
// chamada da função  
somar(2,3);
```

A função a seguir retorna um valor do tipo `number`, por este motivo ela recebeu a anotação de tipo de retorno:

```
function dif(a:number,b:number):number {  
    return a - b;  
}  
  
// chamada da função  
const res = dif(5,3);  
console.log("Resultado:", res);
```

Parâmetro condicional: no TS, é possível definir parâmetros condicionais em uma função utilizando **tipos condicionais**. O tipo condicional é definido colocando `?` após a declaração da variável. Desta forma, a variável terá valor `undefined` se ela não receber valor como parâmetro. No exemplo a seguir a função `dif` pode ser chamada passando um ou dois números como parâmetro, pois o 2º parâmetro é condicional:

```
function dif(a:number,b?:number):number {  
    // o argumento b será undefined se ele não for passado como parâmetro  
    if( b === undefined ){  
        return a;  
    }  
    else{  
        return a - b;  
    }  
}  
  
// os parâmetros a e b possuem números  
console.log("Resultado:", dif(5,3));  
// o parâmetro b terá valor undefined  
console.log("Resultado:", dif(5));
```

Tanto no JS como no TS uma função pode ser declarada de três formas:

- 1 Declaração de função nomeada: o nome da função está imediatamente após a palavra-reservada `function`. No exemplo a seguir `dif` é o nome da função, ou seja, para chamarmos a função usamos o seu nome seguido pelos parênteses, como exemplo, `dif(5,3)`:

```
function dif(a:number,b:number):number{  
    return a - b;  
}
```

- 2 Declaração de função anônima: a função não possui nome. Nesse caso, usamos uma “expressão de função” para atribuir uma função anônima a uma variável. No exemplo a seguir a variável `somar` recebeu a função anônima, ou seja, para chamarmos a função anônima usamos o nome da variável seguida pelos parênteses, como exemplo, `somar(2,3)`:

```
const somar = function(a:number,b:number):number{
    return a + b;
}
```

- 3 Declaração de arrow function (função seta/flecha): possui uma sintaxe mais curta quando comparada com a função anônima. Arrow functions são sempre anônimas. A seguir tem-se três declarações distintas de arrow function:

```
const mult = (a: number, b: number):number => {
    return a * b;
};
const div = (a: number, b: number):number => a / b;
const pow = (a: number, b: number):number => { return a ** b };
```

Quando a arrow function possui no corpo apenas a instrução `return`, então podemos retirar o `return` e as chaves, assim como fizemos na função `div`. Mas se adicionarmos as chaves, como fizemos na função `pow`, então a função precisará ter a instrução `return`.

XIII. Export e import

No contexto do TS um módulo é um arquivo que exporta alguma entidade (variável, função, classe, interface ou tipo). Os pacotes são pastas, na estrutura de arquivos do projeto, que possuem algum módulo que exporta alguma entidade.

A exportação e importação é um recurso importante na modularização e reutilização de código. Dado que o código pode ser organizado em pastas e módulos e consumidos em diferentes partes do programa.

A seguir tem-se três formas de exportar e importar entidades. Considere nas explicações o pacote `operacoes` com os módulos `matematica`, `saudacao` e `texto`.

Estrutura de pastas e arquivos:	Módulo matematica: exemplo de exportação por default
<div> <div>EXEMPLO</div> <div> <div>> node_modules</div> <div> <div>> src</div> <div> <div>> operacoes</div> <div> <div>TS matematica.ts</div> <div>TS saudacao.ts</div> <div>TS texto.ts</div> <div>TS index.ts</div> <div>{ } package-lock.json</div> <div>{ } package.json</div> <div>TS tsconfig.json</div> </div> </div> </div> </div> </div>	<pre>export default function somar(x:number, y:number):number { return x + y; }</pre>
	Módulo texto: exemplo de exportação nomeada
	<pre>export function concatenar(x:string, y:string):string { return x + y; }</pre>
	<pre>export const carro = "Uno";</pre>
	Módulo saudacao: exemplo de exportação agrupada
	<pre>function msg(): void { console.log("olá"); } function resposta(): void { console.log("Boa noite"); }</pre>

```
export { msg, resposta };
```

Módulo index:

```
import somar from "./operacoes/matematica";
import add from "./operacoes/matematica";
import { concatenar, carro } from "./operacoes/texto";
import { msg, resposta } from "./operacoes/saudacao";

console.log(add(5,11));
console.log(somar(2,3));
console.log(concatenar("o","i"));
console.log(carro);
msg();
resposta();
```

1. Exportação por padrão: um arquivo pode ter apenas uma entidade exportada por padrão. Usa-se o termo `export default` antes da entidade a ser exportada. O módulo `matematica` exporta por default a função `somar`.

A importação de uma entidade exportada por default pode ter qualquer nome na importação. No exemplo a seguir o mesmo recurso foi chamado de `somar` e `add`:

```
import somar from "./operacoes/matematica";
import add from "./operacoes/matematica";
```

2. Exportação nomeada: o termo `export` vem antes de cada entidade exportada. O módulo `texto` exporta a função `concatenar` e a variável `carro`.

A importação requer que o recurso importado tenha o nome exato da exportação e seja desestruturado, ou seja, fique entre chaves. Os recursos podem ser importados agrupados:

```
import { concatenar, carro } from "./operacoes/texto";
```

ou podem ser importados separadamente:

```
import { concatenar } from "./operacoes/texto";
import { carro } from "./operacoes/texto";
```

3. Exportação agrupada: as entidades exportadas são estruturadas em um objeto JSON. O módulo `saudacao` exporta as funções `msg` e `resposta`.

A importação requer que o recurso importado tenha o nome exato da exportação e seja desestruturado, ou seja, fique entre chaves. Os recursos podem ser importados agrupados:

```
import { msg, resposta } from "./operacoes/saudacao";
```

ou podem ser importados separadamente:

```
import { resposta } from "./operacoes/saudacao";
import { msg } from "./operacoes/saudacao";
```

Um módulo pode ter somente um recurso exportado por default (por padrão), mas pode ter vários exportados de forma nomeada ou agrupada.

Exercícios

Veja o vídeo se tiver dúvidas nos exercícios: <https://youtu.be/12Ocdrdcxwc>

Instruções para criar o projeto e fazer os exercícios:

1. Crie a pasta `aula1` (ou qualquer outro nome sem caracteres especiais) no local de sua preferência do computador;
2. Abra a pasta no VS Code e acesse o terminal do VS Code;
3. No terminal, execute o comando `npm init -y` para criar o arquivo `package.json`;
4. No terminal, execute o comando `npm i -D ts-node typescript` para instalar os pacotes `ts-node` e `typescript` como dependências de desenvolvimento;
5. Crie uma pasta de nome `src` na raiz do projeto e crie os arquivos dos exercícios nela – assim como é mostrado na figura ao lado;
6. Adicione na propriedade `scripts` do `package.json` os comandos para executar cada exercício. Como exemplo, aqui estão os comandos para executar os dois primeiros exercícios:



```
{
  "name": "aula1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "um": "ts-node ./src/exercicio1",
    "dois": "ts-node ./src/exercicio2"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^10.9.1",
    "typescript": "^5.1.3"
  }
}
```

Observação: se o comando `ts-node` não funcionar, então use o programa `npx` para executar o comando `ts-node`:

```
"scripts": {
  "um": "npx ts-node ./src/exercicio1",
  "dois": "npx ts-node ./src/exercicio2"
},
```

Exercício 1: Adicionar as anotações de tipo nas variáveis e no retorno da função.

A função `somar` recebe dois números e retorna a soma deles ou eles concatenados como string.

```
function somar(a,b) {  
    if( a < b ){  
        return a + b; //somar  
    }  
    else{  
        return "" + a + b; //concatenar  
    }  
}
```

```
console.log("Resultado:", somar(1,2));  
console.log("Resultado:", somar(5,3));
```

Exemplo de saída:

```
PS D:\aula1> npm run um  
> aula1@1.0.0 um  
> ts-node ./src/exercicio1  
Resultado: 3  
Resultado: 53
```

Exercício 2: Adicionar as anotações de tipo nas variáveis e no retorno da função.

A função `converter` recebe um array de números e retorna o array com os elementos convertidos para string.

```
function converter(vet){  
    const res = [];  
    for(let i = 0; i < vet.length; i++){  
        res[i] = "" + vet[i];  
    }  
    return res;  
}  
  
const vetor = [5,3,1,8,2];  
console.log("Resultado:", converter(vetor));
```

Exemplo de saída:

```
PS D:\aula1> npm run dois  
> aula1@1.0.0 dois  
> ts-node ./src/exercicio2  
Resultado: [ '5', '3', '1', '8', '2' ]
```

Exercício 3: Adicionar os tipos nas variáveis e no retorno da função usando a notação de tipos genéricos (generics).

A função `operar` recebe um array de números e um array de strings como parâmetro e retorna um array com os elementos dos arrays concatenados.

```
function operar(v1,v2) {  
    let res = [];  
    for(let i = 0; i < v1.length; i++){  
        res[i] = v1[i] + v2[i];  
    }  
    return res;  
}
```

Exemplo de saída:

```
PS D:\aula1> npm run tres  
> aula1@1.0.0 tres  
> ts-node ./src/exercicio3  
Resultado: [ '5M', '3a', '1r', '8i', '2a' ]
```

```
const vet1 = [5,3,1,8,2];
const vet2 = ["M","a","r","i","a"];
console.log("Resultado:", operar(vet1,vet2));
```

Exercício 4: Reescrever a função do Exercício 3 como função anônima.

Observação: o nome da função e das variáveis precisarão ser alterados no arquivo exercicio4.ts, pois ocorrerá a seguinte mensagem de erro ao digitar o código: "Cannot redeclare block-scoped variable 'operar'". O erro ocorre pelo fato de os arquivos exercicio3.ts e exercicio4.ts estarem dentro da mesma pasta.

Exercício 5: Reescrever a função do Exercício 3 na notação de Arrow Function.

Exercício 6: Reescrever a função isPar usando a notação de Arrow Function.

Restrição: você não poderá usar chaves e o termo return.

```
function isPar(nro) {
    return nro % 2 == 0 ? true : false;
}
```

```
console.log("Resultado:", isPar(2));
console.log("Resultado:", isPar(3));
```

Exemplo de saída:

```
PS D:\aula1> npm run seis
> aula1@1.0.0 seis
> ts-node ./src/exercicio6
Resultado: true
Resultado: false
```

Exercício 7: Adicionar os tipos de dados nas lacunas do código a seguir.

Observe que o JS e TS possuem o tipo de dado Function.

Exemplo de saída:

```
PS D:\aula1> npm run sete
> aula1@1.0.0 sete
> ts-node ./src/exercicio7
5 é ímpar
3 é ímpar
1 é ímpar
8 é par
2 é par
```

```
const parImpar:Function = (nro:_____):_____ => nro % 2 == 0 ? "par" : "ímpar";

const somarArray:_____ = (v:_____):_____ => {
    for(let i = 0; i < v.length; i++){
        console.log(v[i], "é", parImpar(v[i]));
    }
}

const v:_____ = [5, 3, 1, 8, 2];
somarArray(v);
```

Exercício 8: Adicionar os tipos de dados no código a seguir.

```
// gera um número inteiro aleatório no intervalo [0,99]
const aleatorio = () => Math.floor(Math.random() * 100);

// gera um array de números aleatórios
const aleatorios = (quantidade) => {
  const res = [];
  for(let i = 0; i < quantidade; i++){
    // obtém um número aleatório e coloca no final do array
    res.push(aleatorio());
  }
  return res;
}

console.log("5 números aleatórios:", aleatorios(5));
console.log("3 números aleatórios:", aleatorios(3));
```

Exemplo de saída:

```
PS D:\aula1> npm run oito
> aula1@1.0.0 oito
> ts-node ./src/exercicio8

5 números aleatórios: [ 3, 19, 16, 90, 0 ]
3 números aleatórios: [ 46, 37, 39 ]
```

Exercício 9: Adicionar os tipos de dados no código a seguir.

```
// substitui os termos de busca pelo novo caractere no texto de entrada
const substituir = function(entrada, letra, novo) {
  let res = [];
  for(let i = 0; i < entrada.length; i++){
    if( entrada[i] == letra ){
      res.push(novo);
    }
    else{
      res.push(entrada[i])
    }
  }
  return res;
};
```

Exemplo de saída:

```
PS D:\aula1> npm run nove
> aula1@1.0.0 nove
> ts-node ./src/exercicio9

Substituir 'a' por '-': [
  'M', '-', 'r',
  'i', '-', 'n',
  '-'
]
Substituir 'e' por '8': [
  'P', 'e', 'r', 's',
  'p', 'e', 'c', 't',
  'i', 'v', '-'
]
```

```
console.log("Substituir 'a' por '-':", substituir("Mariana","a","-"));
console.log("Substituir 'e' por '8':", substituir("Perspectiva","a","-"));
```

Exercício 10: Adicionar os tipos de dados no código a seguir.

Exemplo de saída:

```
PS D:\aula1> npm run dez
> aula1@1.0.0 dez
> ts-node ./src/exercicio10
5 + 3: 8
5 - 3: 2
```

```
const sum = (a,b) => a + b;

const dif = (a,b) => a - b;

//uma função pode receber outra função como parâmetro
const operacao = (f,a,b) => f(a,b);

console.log("5 + 3:", operacao(sum,5,3));
console.log("5 - 3:", operacao(dif,5,3));
```