**Formação TypeScript**

**TypeScript Parte I – Evoluindo seu JavaScript**

1. **Aula 1 – Porque usar TypeScript:**
   1. A primeira coisa quando vamos codar em typescript é fazer a preparação do ambiente, ou seja, instalar node.js e depois dar um npm install no terminal dentro da pasta do projeto para instalar as dependências do server para rodar nossa aplicação.
      1. Npm run server: Starta o server quando rodar o comando no terminal dentro da pasta do projeto.
      2. No meu caso estou usando a extensão live server do VScode pois não funcionou de jeito nenhum.
   2. Negociação, modelagem e regras:
      1. #: variável privada, não se altera.
      2. Criamos uma classe de negociação para nossa aplicação onde possui as variáveis de data, quantidade e valor privadas com um # antes delas e um construtor que atribui esses valores ao ser criado.
      3. Teoricamente não poderíamos conseguir alterar os valores dessas variáveis diretamente, então fizemos o teste com uma atribuição direta fora do construtor.
      4. Ao fazer um console.log do antes e depois dessa atribuição direta, notamos que ao invés de alterar o valor da variável privada, ele criou uma nova variável com o valor novo atribuído:

// Módulo da negociação

export class Negociacao {

  #data;

  #quantidade;

  #valor;

  constructor(data, quantidade, valor) {

    this.#data = data;

    this.#quantidade = quantidade;

    this.#valor = valor;

  }

}

// Módulo do APP

import { Negociacao } from "./models/negociacao.js";

const negociacao = new Negociacao(new Date(), 10, 100);

console.log(negociacao)

negociacao.quantidade = 1000

console.log(negociacao)



* + 1. O módulo do app é o que está vinculado a nossa aplicação web.
  1. Finalização do modelo:
     1. Lembrando que se quiser visualizar uma variável privada, não conseguimos mesmo que tentemos colocar o # antes dela num console.log:

console.log(negociacao.#data)



* + 1. Se tentarmos visualizar sem o # ele dá como Undefined.
    2. Portanto precisamos criar getters para essas propriedades para que possamos visualizar elas, embora ainda não poderemos fazer alterações diretamente utilizando elas, pois são getters e não setters.
  1. Motivação do TypeScript:
     1. Quando utilizamos JS só vamos descobrir que cometemos um erro no código em run time, ou seja, em tempo de execução dele, seja já na produção ou então no ambiente de testes, tendo que fazer o código voltar para revisar o que deu errado.
     2. O TS trás tudo o que o JS possuí e ainda muito mais, fazendo com que esse tipo de erro não aconteça. O tipo de coding é igual, mas com muito mais ferramentas pois ela é um super do ECMA desenvolvido pela Microsoft. Uma das funcionalidades incríveis é que conseguimos identificar e pegar esses erros em tempo de desenvolvimento, adiantando muito caso algo dê errado.
  2. O que aprendemos:
     1. Introdução ao projeto e sua estrutura;
     2. Um pouco sobre módulos do ECMASCRIPT;
     3. Modelagem de uma Negociação em Javascript;
     4. Buracos em nossa modelagem por limitações da linguagem Javascript.

1. **Aula 2 – TypeScript e Compilador:**
   1. Instalando o TypeScript:
      1. npm install typescript --save-dev: Instalação da versão mais recente do TS via terminal no VScode.
      2. npm install typescript@n.n.n --save-dev: Instalação de uma versão específica do TS via terminal no VScode. Substituir ‘n’ pelos números da versão que quiser.
   2. Arquivos TS:
      1. A extensão de arquivos com código em TS é justamente essa: .ts.
      2. Simplesmente ao alterar a extensão do arquivo, antes mesmo de instalar o compilador e fazer a configuração do TS, ele já começa a nos mostrar os erros do nosso código e, ao colocar o mouse em cima do erro, ele exibe o motivo de estar errado.
      3. Se tentarmos carregar esse arquivo no navegador ele não irá reconhecer, pois o navegador nem sabe o que é o TS.
      4. Por causa disso temos o compilador e uma outra pasta chamada app. Na pasta chamada app nós deixamos tudo da nossa aplicação que será escrito em TS e, na pasta dist, onde estavam nossos arquivos que antes eram JS, ficará tudo o que o navegador consegue ler.
      5. O compilador serve justamente para converter tudo o que vamos escrever em TS para JS e jogar automaticamente nas pastas correspondentes dentro da pasta dist.
   3. **SOLUÇÃO PARA O PROBLEMA DE NÃO RODAR O COMPILADOR E NEM O SERVER!!!!!**
      1. Não estava nada funcionando, nem o compilador nem o server, não permitindo com que eu desse continuação no curso.
      2. O que eu fiz para que funcionasse foi deletar a pasta de node\_modules e o arquivo package-lock.json de dentro da pasta do projeto e baixar a versão mais atualizada do nodejs via chocolatey diretamente na máquina.
      3. Ele provavelmente será instalado na pasta C:\Users\nomeDoUsuário\AppData\Roaming\, onde você encontrará a seguir diversas pastas, dentre elas a do npm.
      4. Ao entrar no npm\node\_modules, você deverá encontrar 2 pastas ‘npm’ e ‘lite-server’, se a segunda não estiver, provavelmente deu erro durante a instalação do gerador/conector com o servidor, para corrigir basta pesquisar por lite-server no google, mas, se bem me lembro, você pode instalar ele globalmente a partir do npm install --global lite-server, ou dentro do próprio projeto a partir do npm install lite-server –save-dev. Na segunda opção deverá colocar “scripts”: {“dev”: “lite-server”} dentro do package.json e rodar um npm run dev no console na pasta do projeto para que inicie o server. Se o index.html que exibirá sua aplicação não estiver na raiz do projeto, coloque o parâmetro –baseDir=caminho\da\pasta\do\index\. Lembrando que tudo o que foi dito dentro da segunda opção é para ser feito na pasta do projeto.
      5. Ao instalar globalmente a pasta do lite-server deverá aparecer e, a partir desse ponto, iniciar o server normalmente no seu projeto.
      6. Para começarmos com o TS primeiro precisamos instalar ele e, para isso, vamos seguir para a pasta no caminho C:\Users\nomeDoUsuário\AppData\Roaming\npm\node\_modules\npm e nessa pasta instalar o typescript na versão que desejar a partir do npm install [typescript@4.2.2](mailto:typescript@4.2.2) (essa versão foi a que usei para o curso, mas sugere utilizar a mais recente. Nesse caso não precisa do @ e nem nada depois dele). Ao rodar esse comando nessa pasta, toda a instalação será feita adequadamente.
      7. Porém, o compilador (tsc (typescript compiler)) provavelmente ainda não funcionará. Para que ele passe a ser reconhecido como um comando precisamos colocar a pasta em que ele se encontra na nossa variável PATH no PowerShell ou CMD. Em ambos os casos para fazer isso precisamos ir lá na pasta onde os comandos do TS se encontram, geralmente sendo no C:\Users\nomeDoUsuario\AppData\Roaming\npm\node\_modules\npm\node\_modules\.bin, mas que para descobrir pode simplesmente usar o comando ls -Recurse \*tsc\* para listar todos os arquivos que possuem essas letras no nome. O que procuramos é uma tríade de tsc, tsc.cmd e tsc.ps1. Após encontrar e entrar na pasta em que esses arquivos estão, podemos usar o comando pwd para ver o caminho todo desde o disco e copiar esse caminho. Em seguida utilizaremos o comando setx PATH “%PATH%;C:\todo\o\camiho\descoberto\pelo\pwd”. Após sua execução aparecerá uma mensagem de êxito e então devemos reiniciar o PS e/ou CMD e também o VScode, ou seja lá a IDE que esteja utilizando, só então veremos o resultado.
         1. **[NOTA!!]:** **Não tenho certeza se no PS realmente precisa ou sequer funciona colocar o %PATH%; e depois todo o caminho para setar o novo caminho nele. Talvez seja necessário colocar o $env:Path ou então só $Path ou $PATH para que dê certo. Se estiver utilizando o CMD, NÃO ESQUEÇA DE COLOCAR DE MANEIRA NENHUMA OU PODE DAR MUITO ERRADO!!!! Para ter certeza que vai ficar tudo bem, faça um backup do path utilizando o echo %PATH% $Path ou $env:Path e guarde o arquivo em segurança para poder recuperar caso de errado. Qualquer dúvida consultar as anotações de CMD/PROMPT. No meu caso eu estava usando o PS e esqueci que precisava colocar o “%PATH%; caminho\novo” para adicionar ao path e não substituir ele todo, mas acabou que o PS identificou que eu queria adicionar e não substituir, então deu certo, mas não confie, faça certo.**
      8. A partir desse ponto tanto o compilador quanto o server init devem estar funcionando adequadamente no seu projeto e com as configurações de tsconfig.json que você fizer para ele.
   4. Configuração básica do compilador:
      1. Para configurar nosso compilador precisamos criar um arquivo chamado tsconfig.json e, dentro dele, colocar algumas pequenas opções como:

{

  "compilerOptions": {        // Define as opções de compilação

    "outDir": "dist/js",      // Diz qual é o diretório em que os arquivos compilados serão direcionados

    "target": "ES6"           // Diz que o alvo de conversão, ou seja, para qual tipo de JS ou ES ele deve converter nossos arquivos TS ao compilar, nesse caso dissemos que queremos o EcmaScript 6

  },

  "include": ["app/\*\*/\*"]     // Diz para o compilador que, ao compilar, é para incluir todos os arquivos que estiverem dentro da pasta app e suas subpastas na sua conversão

}

* + 1. Após configurar devemos ir até a pasta package.json do nosso projeto e colocar dentro do script um comando “compile”: “tsc” e dentro das “devDepencencies”: {“typescript”: “^4.2.2”}.
       1. Coloquei 4.2.2 pois foi a versão do TS que instalei, caso tenha outra, verifique a versão e coloque-a aqui.
  1. Aprimorando a configuração:
     1. Adicionamos uma configuração que não permite a compilação e geração dos arquivos JS se nosso script de TS estiver com erros:

    "noEmitOnError": true     // Não permite gerar arquivos JS enquanto tiver erros no nosso código

* 1. Automatizando a compilação de arquivos:
     1. Ficar finalizando nosso server, rodando a compilação e depois rodar o server novamente para ver o resultado é bem ruim, por isso colocamos uma nova função no nosso script no package.json chamada watch e que recebe tsc -w, ou seja, o compilador fica assistindo a qualquer mudança existente no nosso TS e, ao notar alguma diferença quando salvamos o arquivo, ele compila automaticamente gerando os novos arquivos js.

    "watch": "tsc -w"

* 1. O modificador private:
     1. O # é o método mais recente de definir que algo é privado no JS, contudo, não no TS. Para definir que uma propriedade é privada em TS precisamos colocar o \_propriedade em todas elas.
     2. Entretanto, ao tentar acessar de fora do construtor ainda será possível, mesmo que esteja com o \_. Para corrigir isso, colocamos o *private* antes da \_propriedade na hora de definir ela, assim, sempre que tentarmos alterar de fora do construtor, teremos um erro como estava sendo com a # anteriormente:

private \_data;

    private \_quantidade;

    private \_valor;

* + 1. Ao ver o JS notamos que todas as nossas propriedades privadas possuem somente o \_ antes delas, o que não garante segurança nenhuma para o nosso código e que as pessoas que acessarem a aplicação modifique essas propriedades diretamente.
    2. O que garantimos com esse método de privatização é que ninguém será capaz de alterar nossas variáveis diretamente em tempo de coding e compilação.
  1. O que aprendemos:
     1. Download do TypeScript;
     2. Configuração do compilador e papel do tsconfig.json;
     3. Integração com scripts do Node.js;
     4. Modificadores de acesso private e public;
     5. Benefícios iniciais da linguagem TypeScript.

1. **Aula 3 – Benefícios da Tipagem Estática:**
   1. O controller de negociação:
      1. Criamos uma classe que recebe das as variáveis da nossa aplicação e criamos um método para ela que sempre que adicionar ela irá imprimir o valor colocado. Isso é provisório, apenas para testar se está funcionando:

export class NegociacaoController {

  private inputData;

  private inputQuantidade;

  private inputValor;

  constructor() {

    this.inputData = document.querySelector('#data');

    this.inputQuantidade = document.querySelector('#quantidade');

    this.inputValor = document.querySelector('#valor');

  }

  adiciona() {

    console.log(this.inputData);

    console.log(this.inputQuantidade);

    console.log(this.inputValor);

  }

}

* + 1. O controller é o que fará o intermédio, é ele que irá mandar os dados para criar a instância de negociação quando clicar no botão incluir da nossa aplicação.
  1. Integração com o formulário:
     1. Importamos nosso arquivo de negociação-controller e pegamos com querySelector o formulário da nossa página.
     2. Criamos um addEventListener() para que sempre que for submetido o controller use o método adiciona, imprimindo o que foi escrito no console:

import { NegociacaoController } from "./controllers/negociacao-controller.js";

const controller = new NegociacaoController();

const form = document.querySelector('.form');

form.addEventListener('submit', event => {

  event.preventDefault();

  controller.adiciona();

});

* + 1. O event.preventDefault() faz com que a página não recarregue ao submeter o formulário.
  1. Surpresa ao instanciar uma negociação:
     1. Fizemos o código correto agora, ao invés de exibir o que o usuário colocou, instanciamos o nosso objeto:

adiciona() {

    const negociacao = new Negociacao(

      this.inputData.value,

      this.inputQuantidade.value,

      this.inputValor.value

    );

    console.log(negociacao)

  }

* + 1. Pegamos somente o valor de cada input, caso contrário seria exibido a tag inteira no console.log.
    2. Porém, mesmo que tenha dado certo quando executamos nossa aplicação web, ela também deu errado, isso porque todos os valores recebidos estão com formato de str e não de data e int/float como deveria.
    3. Apesar de termos esse erro o TS não mostrou pra gente na hora de compilar. Isso acontece porque, como o nome diz, TS é uma linguagem de tipagem estática e podemos dizer para ele qual é o tipo de cada propriedade que queremos.
    4. Ao fazer isso ele passará a nos mostrar que o tipo esperado em um construtor, por exemplo, é diferente do que ele receberá, dando um erro de compilação e nos permitindo concertar antes do run time.
  1. O tipo implícito any:
     1. Como estamos trabalhando com TS, não faz sentido deixar as propriedades com valor any, uma vez que não estaríamos utilizando o type do typescript.
     2. Considerando isso, podemos adicionar uma configuração no nosso tsconfig.json que impede que possamos deixar as propriedades como any, sempre tendo que definir qual é o tipo dela:

    "noImplicitAny": true     // Não permite que deixemos as nossas propriedades como any. Temos que sempre definir um tipo para elas.

* + 1. Para definir o tipo de uma propriedade, colocamos :tipo:

constructor(data: Date, quantidade: number, valor: number) {

        this.\_data = data;

        this.\_quantidade = quantidade;

        this.\_valor = valor;

    }

* + 1. O interessante é colocarmos também na definição da nossa propriedade, lá no início:

export class Negociacao {

    private \_data: Date;

    private \_quantidade: number;

    private \_valor: number;

* 1. Ajustando nosso controller:
     1. Podemos explicitar que queremos que o tipo da nossa propriedade seja any, seguindo o que foi mostrado acima.
     2. Entretanto não é muito bom fazer isso, pois como o TS trabalha com tipagem definida, é muito melhor e mais fácil de codar quando passamos um tipo definido para ele.
     3. Um exemplo disso é a ajuda do autocomplete, ao tentar utilizar o autocomplete em uma propriedade do tipo any, ele não mostra nada, pois o TS não sabe o que você quer com aquela propriedade.
     4. Em contra partida, após definir o tipo da nossa variável para data, por exemplo, o autocomplete irá exibir todos os métodos disponíveis que podemos utilizar com propriedades do tipo data. O mesmo para number e afins.
     5. Além desses tipos mais comuns, o TS também possuí tipos para elementos do DOM, como o HTMLInputElement, e podemos utilizar eles para tipar nossas propriedades.

export class NegociacaoController {

  private inputData: HTMLInputElement;

  private inputQuantidade: HTMLInputElement;

  private inputValor: HTMLInputElement

* + 1. Entretanto, ao fazer essa tipagem, nosso compilador exibirá um erro dizendo que o que estamos tentando receber é desses inputs são strings, mas que deveríamos estar recebendo datas ou números, que foi o que definimos no negociação.ts:



* + 1. Só para deixar claro:
       1. Código do negociacao.ts:

export class Negociacao {

    private \_data: Date;

    private \_quantidade: number;

    private \_valor: number;

    constructor(data: Date, quantidade: number, valor: number) {

        this.\_data = data;

        this.\_quantidade = quantidade;

        this.\_valor = valor;

    }

    get data() {

        return this.\_data;

    }

    get quantidade() {

        return this.\_quantidade;

    }

    get valor() {

        return this.\_valor;

    }

    get volume() {

        return this.\_quantidade \* this.\_valor;

    }

}

* + - 1. Código do negociacao-controller.ts:

import { Negociacao } from "../models/negociacao.js";

export class NegociacaoController {

  private inputData: HTMLInputElement;

  private inputQuantidade: HTMLInputElement;

  private inputValor: HTMLInputElement;

  constructor() {

    this.inputData = document.querySelector('#data');

    this.inputQuantidade = document.querySelector('#quantidade');

    this.inputValor = document.querySelector('#valor');

  }

  adiciona() {

    const negociacao = new Negociacao(

      this.inputData.value,

      this.inputQuantidade.value,

      this.inputValor.value

    );

    console.log(negociacao)

  }

}

* 1. Convertendo dados de entrada:
     1. Fazemos conversões utilizando o parseInt(this.inputHtml.value) para números inteiros e parseFloat(this.inputHtml.value) para números com ‘.’.
     2. No caso da data, nós recebemos o ‘–‘ separando ano, mês e dia do nosso input. Para fazer a separação por ‘,’ criamos uma regex e utilizamos o replace, substituindo tudo o que o regex encontrar por ‘,’:

const exp = /-/g;

    const date = new Date(this.inputData.value.replace(exp, ','));

* + 1. Precisamos fazer essa substituição porque para que um objeto do tipo date seja criado ele precisa receber uma string com as separações por ‘,’, tipo 11,11,2022 e, o que recebemos do input vem no formato 11-11-2022.
    2. Para facilitar fizemos todas as conversões e guardamos em variáveis, passando para o nosso construtor somente elas ao invés de todo aquele this.InputHtml.value:

  adiciona() {

    const exp = /-/g;

    const date = new Date(this.inputData.value.replace(exp, ','));

    const quantidade = parseInt(this.inputQuantidade.value);

    const valor = parseFloat(this.inputValor.value);

    const negociacao = new Negociacao(date, quantidade, valor);

    console.log(negociacao)

  }

* 1. Organizando melhor nosso código:
     1. Além das propriedades, os métodos de classe também são tipados e, sempre que esperarmos um retorno é bom tipar qual retorno esperamos:
        1. Código negociacao.ts:

export class Negociacao {

    private \_data: Date;

    private \_quantidade: number;

    private \_valor: number;

    constructor(data: Date, quantidade: number, valor: number) {

        this.\_data = data;

        this.\_quantidade = quantidade;

        this.\_valor = valor;

    }

    get data(): Date {

        return this.\_data;

    }

    get quantidade(): number {

        return this.\_quantidade;

    }

    get valor(): number {

        return this.\_valor;

    }

    get volume(): number {

        return this.\_quantidade \* this.\_valor;

    }

}

* + - 1. Código negociacao-controller.ts:

import { Negociacao } from "../models/negociacao.js";

export class NegociacaoController {

  private inputData: HTMLInputElement;

  private inputQuantidade: HTMLInputElement;

  private inputValor: HTMLInputElement;

  constructor() {

    this.inputData = document.querySelector('#data');

    this.inputQuantidade = document.querySelector('#quantidade');

    this.inputValor = document.querySelector('#valor');

  }

  adiciona(): void {

    const negociacao = this.criaNegociacao();

    console.log(negociacao)

  }

  criaNegociacao(): Negociacao {

    const exp = /-/g;

    const date = new Date(this.inputData.value.replace(exp, ','));

    const quantidade = parseInt(this.inputQuantidade.value);

    const valor = parseFloat(this.inputValor.value);

    return new Negociacao(date, quantidade, valor);

  }

}

* + 1. Sempre tipar os getters também. Se esperamos que o getter nos retorne um número, colocamos isso como tipo, se esperamos que o nosso método de classe retorne um construtor, devemos colocar o nome do construtor na tipagem, se esperamos que nosso método não retorne nada, colocamos void.
    2. É uma boa prática colocar o tipo de retorno esperado, se houver, logo na hora da criação do método, pois dessa forma garantimos que durante a sua escrita faremos tudo corretamente, já que o TS não deixará nada errado lá dentro.
    3. Criamos um método que cria a negociação e separamos do que apenas adiciona ela para deixar mais organizado.
  1. Limpando o formulário:
     1. Para limpar o formulário basta atribuir o valor ‘’ em todos os inputs.
     2. Decidimos voltar o foco para o primeiro input logo após a inclusão da nova proposta, para isso utilizamos o .focus.
     3. O nosso método não retorna nada, então já colocamos o tipo como void.
     4. Como queremos que tudo isso ocorro depois de ter sido adicionado, colocamos esse método como último do método adiciona:

adiciona(): void {

    const negociacao = this.criaNegociacao();

    console.log(negociacao);

    this.limparFormulario();

  }

  criaNegociacao(): Negociacao {

    const exp = /-/g;

    const date = new Date(this.inputData.value.replace(exp, ','));

    const quantidade = parseInt(this.inputQuantidade.value);

    const valor = parseFloat(this.inputValor.value);

    return new Negociacao(date, quantidade, valor);

  }

  limparFormulario(): void {

    this.inputData.value = '';

    this.inputQuantidade.value = '';

    this.inputValor.value = '';

    this.inputData.focus();

  }

* 1. O que aprendemos:
     1. O tipo implícito any;
     2. Benefícios da tipagem estática;
     3. Mais configurações do compilador tsc;
     4. Retorno de método explícito;
     5. Conversão de valores da interface do usuário.

1. **Aula 4 – Avançando na Modelagem do Dommínio:**
   1. Modelo de uma lista de negociações:
      1. Começamos a desenvolver um modelo de lista de negociações, onde receberá todas as negociações incluídas através do modelo de negociação e terá vários métodos para poder exibir as negociações feitas e afins, mas não será possível apagar uma negociação como está sendo no momento.
      2. Afinal, não faz sentido nenhum criarmos uma negociação em um dia e no outro ir lá e apagar ela criando uma nova como se nada tivesse acontecido. As nossas negociações devem ter um histórico inalterável.
      3. O modelo que estamos criando terá um array privado para armazenar as negociações feitas pelo modelo negociação.
      4. Ao criar esse array em propriedade privada temos como resultado um erro de tipagem. O TS nos diz que essa propriedade possuí um array do tipo any implícito. Veremos como corrigir a tipagem.
   2. Primeiro contato com Generics:
      1. Quando criamos um array ele é sempre do tipo genérico, ou seja, any, pois assim ela pode receber qualquer tipo de valor.
      2. Entretanto, não é isso que queremos e nem mesmo o TS permite que isso seja feito, já que colocamos aquela configuração do “noImplicitAny” = true, não permitindo que deixemos nada com tipagem any implícita.
      3. No nosso caso queremos que nossa array receba as negociações incluídas pelo modelo de negociação, então a tipagem do nosso array vai ser:

import { Negociacao } from "./negociacao.js";

export class Negociacoes {

  private negociacoes: Array<Negociacao> = [];

}

* + 1. Para definir a tipagem de um array precisamos colocar o diamante ‘<>’ na frente e dizer qual é o tipo desse array, ou seja, o que esperamos que seja adicionado dentro dele.
    2. Ao colocar a tipagem como um modelo que criamos, ele automaticamente faz a importação para o arquivo.
    3. Sempre precisamos lembrar de adicionar o .js no final da importação, pois ele não faz isso sozinho.
    4. Mesmo que estejamos trabalhando com arquivos ts, nossa importação deve ser js.
    5. Esse tipo de tipagem de array é interessante pois 99% das vezes nós queremos que tenha somente um tipo de dados nesse array e, quando vamos fazer uma iteração por exemplo, o elemento que criamos para fazer a iteração será automaticamente definido com a mesma tipagem do array, pois o TS entende que nada que está naquela lista pode ser diferente daquele tipo.
    6. Para exemplificar melhor criamos uma iteração:

const list: Array<string> = [];

list.push('Bruno');

list.push('Henrique');

for(let nome of list) {

  nome.

}

* + 1. No código acima dissemos que nosso array deve ser uma string e adicionamos através do push dois elementos de string nele. Considerando isso, o typescript entende que, por convenção, o let nome será também uma string, nos mostrando no autocomplete todos os métodos que podemos utilizar com strings quando colocamos o ‘.’ na frente dele:



* + 1. Não só isso como também, ao colocar o mouse em cima do ‘let nome’, ele nos mostra que o tipo dessa variável é string:



* + 1. Se alterarmos o tipo do array, mas não alterar o que está sendo adicionado através do push teremos um erro de compilação:

const list: Array<number> = [];

list.push('Bruno');

list.push('Henrique');

for(let nome of list) {

  nome.

}



* + 1. E automaticamente o nosso ‘let nome’ se torna do tipo number juntamente com nosso array, pois definimos que esse seria o tipo dele. E, assim como no caso das strings, o autocomplete dele exibirá todos os métodos que podem ser utilizados com o tipo number:





* + 1. Claro que podemos também definir explicitamente que o tipo do nosso array será any, assim não teremos nenhum erro de compilação por any implícito e nosso array poderá receber todo tipo de elemento, não dando erro de compilação se tentar dar um push de str, number ou até mesmo Date.
  1. Avançando com nosso modelo:
     1. Criamos o método de classe que adiciona a negociação na lista privada de negociações e o método lista que vai listar todas as negociações dentro do nosso array:

import { Negociacao } from "./negociacao.js";

export class Negociacoes {

  private negociacoes: Array<Negociacao> = [];

  adiciona(negociacao: Negociacao) {

    this.negociacoes.push(negociacao);

  }

  lista(): Array<Negociacao> {

    return this.negociacoes;

  }

}

* 1. Um problema não esperado:
     1. Fomos para o arquivo de negociacao-controller.ts, fizemos a importação do nosso modelo de negociacoes, criamos uma propriedade que recebe uma nova negociacao e, no método adiciona desse arquivo, falamos para que a negociação incluída na página seja adicionada no nosso array através do método adiciona dele:

import { Negociacao } from "../models/negociacao.js";

import { Negociacoes } from "../models/negociacoes.js";

export class NegociacaoController {

  private inputData: HTMLInputElement;

  private inputQuantidade: HTMLInputElement;

  private inputValor: HTMLInputElement;

  private negociacoes = new Negociacoes();

  adiciona(): void {

    const negociacao = this.criaNegociacao();

    this.negociacoes.adiciona(negociacao);

    console.log(this.negociacoes.lista());

    this.limparFormulario();

  }

* + 1. O console.log agora imprime a nossa lista completa através do método lista que criamos. Esse código é do arquivo negociacoes-controller.ts.
    2. Porém ainda temos um erro. Ao tentar remover um conteúdo da lista utilizando o pop(), ainda conseguimos, mesmo que toda essa operação de criar o modelo negociações para adicionar cada negociação feita na página no array para que não possa ser removido, tenha justamente essa intenção, não permitir que nenhuma negociação seja removida.
  1. Evitando a mutabilidade da lista:
     1. Podemos evitar isso simplesmente fazendo uma nova lista e colocando cada item da nossa lista imutável dentro dela, assim, o que será exibido para o cliente é a segunda lista, então ele pode adicionar e deletar itens a vontade, mas a nossa lista original não foi alterada.
     2. Para criar essa segunda lista com todos os itens da primeira utilizamos o spred operator (...) antes da nossa lista original no retorno do método lista que exibe nossas negociações:

import { Negociacao } from "./negociacao.js";

export class Negociacoes {

  private negociacoes: Array<Negociacao> = [];

  adiciona(negociacao: Negociacao) {

    this.negociacoes.push(negociacao);

  }

  lista(): Array<Negociacao> {

    return [...this.negociacoes];

  }

}

* 1. O modificador readonly:
     1. Ao colocar a palavra Readonly antes da tipagem de Array do método lista() do modelo de negociacoes.ts, ele cria uma lista idêntica ao array normal, mas sem nenhum método que possa alterar ele, ou seja, não existe push(), pop() e nem nada que um array comum possuí que o permite alterar, dando um erro de compilação sempre que o usuário tentar fazer uma alteração no array utilizando esses métodos:
        1. Código de negociacoes.ts:

lista(): ReadonlyArray<Negociacao> {

    return [...this.negociacoes];

  }

* + - 1. Código de negociacao-controller.ts:

  adiciona(): void {

    const negociacao = this.criaNegociacao();

    this.negociacoes.adiciona(negociacao);

    this.negociacoes.lista().pop()

    console.log(this.negociacoes.lista());

    this.limparFormulario();

  }

* + - 1. Erro ao tentar usar o pop() para alterar o array:



* + 1. Esse tipo é super interessante quando queremos retornar uma lista somente leitura.
  1. O que aprendemos:
     1. Modelagem da classe Negociacoes;
     2. Utilização de Generics;
     3. Revisão sobre encapsulamento;
     4. Questões de mutabilidade e como solucioná-la;
     5. O tipo ReadonlyArray;
     6. Adição de negociações em nossa lista.

1. **Aula 5 – Simplificando Nosso Código:**
   1. Declaração no próprio construtor:
      1. Todo o código abaixo:

private \_data: Date;

    private \_quantidade: number;

    private \_valor: number;

    constructor(data: Date, quantidade: number, valor: number) {

        this.\_data = data;

        this.\_quantidade = quantidade;

        this.\_valor = valor;

    }

* + 1. Pode simplesmente ser substituído pelo código abaixo:

constructor(private \_data: Date, private \_quantidade: number, private \_valor: number) {}

* + 1. Ambos dizem a mesma coisa para o typescript, apenas simplificamos fazendo a declaração de todas as variáveis privadas logo no construtor. Em resumo: reduzimos 9 linhas de código para apenas 1.
  1. Arrays e Generics:
     1. Podemos fazer a declaração da tipagem do array um pouco diferente:
        1. Antes:

  private negociacoes: Array<Negociacao> = [];

* + - 1. Depois:

  private negociacoes: Negociacao[] = [];

* + - 1. Não muda muito, mas é um atalho que é permitido.
    1. A mesma lógica acima serve para o ReadonlyArray:
       1. Antes:

  lista(): ReadonlyArray<Negociacao> {}

* + - 1. Depois:

  lista(): readonly Negociacao[] {}

* 1. Mais sobre readonly:
     1. Nosso nosso código de negociacao.ts criamos vários getters para que pudéssemos ler os valores que nossas propriedades privadas possuem para que ninguém possa ser capaz de alterá-la diretamente.
     2. Mas, para deixar nosso código menor ainda, podemos remover todos os getters, tornar as variáveis que antes estavam como private para public, remover todos os ‘\_’ e colocar o readonly antes de sua declaração, dessa forma, ninguém poderá acessar nossas propriedades diretamente e ainda deixamos nosso código menor:
        1. Antes:

export class Negociacao {

    constructor(

        private \_data: Date,

        private \_quantidade: number,

        private \_valor: number

        ) {}

    get data(): Date {

        return this.\_data;

    }

    get quantidade(): number {

        return this.\_quantidade;

    }

    get valor(): number {

        return this.\_valor;

    }

    get volume(): number {

        return this.\_quantidade \* this.\_valor;

    }

}

* + - 1. Depois:

export class Negociacao {

    constructor(

        public readonly data: Date,

        public readonly quantidade: number,

        public readonly valor: number

        ) {}

    get volume(): number {

        return this.quantidade \* this.valor;

    }

}

* + - 1. Só mantemos o do volume pois ele não é uma propriedade e queremos que apareça como uma para sua visualização.
  1. Mais surpresas em nossa modelagem:
     1. O readonly não permite qualquer tipo de atribuição direta em propriedades primitivas, mas, se tivermos uma propriedade que é um objeto, como o Date(), ele pode ser alterado através de seus próprios métodos como o setDate(), alterando o dia da criação da negociação.
     2. Por conta disso precisamos saber um pouco sobre programação defensiva.
  2. Programação defensiva:
     1. Mesmo que utilizemos o getter para a data o problema acima não seria resolvido, pois o getter faz referência direta à nossa data original, podendo ainda ser alterada com os métodos de um Date().
     2. Para resolver o problema precisamos criar uma cópia da data original e exibir ela para o usuário, desse modo ele poderá alterar o quanto quiser, mas a data original em que nossa proposta foi criada se manterá imutável.
     3. Esse é um ótimo método de programação defensiva, evitando com que as nossas propriedades originais sejam alteradas. Foi exatamente a mesma coisa que fizemos com o array antes de colocar o readonly nele:
     4. Fizemos novamente o getter de data e voltamos a passar um atributo privado para ela:

    constructor(

        private \_data: Date,

        public readonly quantidade: number,

        public readonly valor: number

        ) {}

get data(): Date {

        const data = new Date(this.\_data.getTime());

        return data;

    }

* + - 1. Pegamos a hora em milissegundos da data informada para a propriedade privada \_data e formamos uma nova data com o new Date() a partir dela, atribuindo esse valor na constante data, ou seja, basicamente copiamos a exata data passada para o construtor na hora de criar a negociação na aplicação.
  1. O que aprendemos:
     1. Nova maneira de declaração de array com generics;
     2. O tipo ReadonlyArray;
     3. O modificador readonly;
     4. Getters vs propriedades públicas em modo de leitura;
     5. Programação defensiva.

**TypeScript Parte II – Avançando na Linguagem**

1. **Aula 1 – Elaborando Uma Solução de View:**
   1. Criando nossa primeira view:
      1. Para escrever uma string em várias linhas quebrando-as sem problemas utilizamos o ` string `.
      2. Para criar a nossa visualização começamos com um novo arquivo ts chamado negociacoes-view.ts e criamos uma classe com um método template nela.
      3. Declaramos que esse método deve retornar uma string a partir da tipagem dele e dentro do método criamos uma table html. Esse é o código do negociacoes-view.ts até agora:

export class NegociacoesView {

  template(): string {

    return `

    <table class="table table-hover tamble-bordered">

      <thead>

          <tr>

              <th>DATA</th>

              <th>QUANTIDADE</th>

              <th>VALOR</th>

          </tr>

      </thead>

      <tbody>

      </tbody>

    </table>

    `

  }

}

* 1. Integrando nossa view à página:
     1. Para fazer a integração dinâmica com o html precisamos criar uma div vazia com um ID que pode ser capturado pelo nosso código. Essa div será utilizada para que nosso código possa colocar a table html que criamos a partir do método template acima.
     2. Em seguida, ainda no nosso código view, criamos uma propriedade privada do tipo htmlElement e um construtor que recebe como parâmetro um seletor, ele é do tipo string.
     3. No nosso construtor colocamos que a propriedade elemento recebe uma querySelector do seletor passado como parâmetro, isto é, o ID que iremos pegar da div do index.
     4. Depois do método de template criamos um novo método chamado update que irá fazer um innerHtml do método template no elemento. Código do negociacoes-view.ts:

export class NegociacoesView {

  private elemento: HTMLElement;

  constructor(seletor: string) {

    this.elemento = document.querySelector(seletor);

  }

  template(): string {

    return `

    <table class="table table-hover tamble-bordered">

      <thead>

          <tr>

              <th>DATA</th>

              <th>QUANTIDADE</th>

              <th>VALOR</th>

          </tr>

      </thead>

      <tbody>

      </tbody>

    </table>

    `;

  }

  update(): void {

    this.elemento.innerHTML = this.template();

  }

}

* + 1. Para obter esse seletor, isto é, o ID da div em que a tabela será inserida, precisamos ir lá no código de negociacao-controller.ts, criar uma propriedade privada que recebe um new NegociacoesView recebendo o ID da div como parâmetro e, no construtor desse módulo, ou seja, na parte do código que roda toda vez que a página é carregada, precisamos colocar essa nova propriedade privada, que recebe a classe de NegociacoesView, com os métodos de template e update, para que sempre que a página for carregada o template seja gerado e inserido no html através do código acima. Código que foi adicionado no negociacao-controller.ts:

import { NegociacoesView } from "../views/negociacoes-view.js";

export class NegociacaoController {

  private inputData: HTMLInputElement;

  private inputQuantidade: HTMLInputElement;

  private inputValor: HTMLInputElement;

  private negociacoes = new Negociacoes();

  private negociacoesView = new NegociacoesView('#negociacoesView');

  constructor() {

    this.inputData = document.querySelector('#data');

    this.inputQuantidade = document.querySelector('#quantidade');

    this.inputValor = document.querySelector('#valor');

    this.negociacoesView.template();

    this.negociacoesView.update();

  }

* + 1. Ao abrir a página html notamos que o header da tabela já se faz presente:



* 1. Unindo view e modelo:
     1. Basicamente o que fizemos foi integrar a atualização da tabela sempre que uma nova negociação fosse inserida. Para isso, lá no método template do módulo de negociações-view.ts, nós adicionamos uma interpolação com ${} no tbody e dissemos que, a partir de um modelo, dentro dele seria inserido o retorno de um mapeamento da lista() das negociações, isto é, toda vez que uma negociação fosse adicionada ele iria adicionar uma nova linha na tabela com as informações passadas. Template do negociações-view.ts:

  template(model: Negociacoes): string {

    return `

    <table class="table table-hover table-bordered">

      <thead>

          <tr>

              <th>DATA</th>

              <th>QUANTIDADE</th>

              <th>VALOR</th>

          </tr>

      </thead>

      <tbody>

        ${model.lista().map(negociacao => {

          return `

            <tr>

                <td>?</td>

                <td>${negociacao.quantidade}</td>

                <td>${negociacao.valor}</td>

            </tr>

          `

        }).join('')}

      </tbody>

    </table>

    `;

  }

* + 1. O join(‘’) diz que é para pegar as tuplas que o map() cria, converter todos para str, juntar e deixar separado apenas por um espaço vazio, pois se mantivéssemos como tuplas o table não entenderia e daria erro.
    2. Para que funcionasse tivemos que passar um parâmetro para o template de um modelo que seria do tipo negociações, bem como o update para que ele atualize a table quando algo for inserido:

  update(model: Negociacoes): void {

    const template = this.template(model);

    console.log(template)

    this.elemento.innerHTML = template;

  }

* + 1. Como agora precisamos passar um modelo para que o update funcione, lá no código do negociaçao-controller.ts, passamos o parâmetro negociações para esse método do construtor, bem como para o método adiciona dele:

  constructor() {

    this.inputData = document.querySelector('#data');

    this.inputQuantidade = document.querySelector('#quantidade');

    this.inputValor = document.querySelector('#valor');

    this.negociacoesView.update(this.negociacoes);

  }

  adiciona(): void {

    const negociacao = this.criaNegociacao();

    this.negociacoes.adiciona(negociacao);

    console.log(this.negociacoes.lista());

    this.negociacoesView.update(this.negociacoes);

    this.limparFormulario();

  }

* 1. Formatando a data:
     1. Para formatar a data na table utilizamos o Intl.DateTimeFormat(), pois através dessa técnica a data passada para a aplicação será automaticamente formatada para a timezone que o navegador do usuário estiver:

<td>${new Intl.DateTimeFormat().format(negociacao.data)}</td>

* 1. O que aprendemos:
     1. Inspiração no React para criação de templates declarativos e dinâmicos;
     2. Manipulação declarativa do DOM através de template;
     3. Template dinâmico;
     4. Formatação de datas usando Intl.

1. **Aula 2 – Herança e Reaproveitamento de Código:**
   1. Mensagem view:
      1. Criamos basicamente o mesmo código da table num módulo novo chamado de mensagem-view.ts para inserir no html uma mensagem quando uma nova negociação fosse adicionada.
      2. O que mudamos foi somente o retorno do template. Código do mensagem-view.ts:

export class MensagemView {

  private element: HTMLElement;

  constructor (seletor: string) {

    this.element = document.querySelector(seletor)

  }

  template(model: string): string {

    return `

      <p class="alert alert-info">${model}</p>

    `

  }

  update(model: string): void {

    const template = this.template(model);

    this.element.innerHTML = template;

  }

}

* 1. Herança e pequena surpresa:
     1. Para herdar uma classe utilizamos extends NomeDaClasse logo na criação de uma nova classe:

export class NegociacoesView extends View {

* + 1. A partir dessa herança podemos colocar o construtor e as propriedades que várias classes tem em comum lá na classe mãe e deletar essa repetição das classes filhas.
    2. Elementos/propriedades declaradas como privadas na classe mãe não pode ser acessadas pelas classes filhas, porém, se deixar pública, qualquer um teria acesso. Para concertar isso declaramos essas propriedades como protected, onde todas as classes filhas dessa podem usar, mas outras que não são filhas não podem. Código da classe mãe view:

export class View {

  protected elemento: HTMLElement;

  constructor(seletor:string) {

    this.elemento = document.querySelector(seletor);

  }

}

* + 1. A partir daqui podemos deletar todo esse código que também está nas classes filhas.
  1. Mais surpresas com Herança:
     1. Quando tentamos criar um código genérico, acaba que não da certo por causa da tipagem. Um mesmo método da classe mãe tipada como string deveria ser herdada e usada para 2 classes filhas, mas com tipagens diferentes. Na que possui tipagem string, funciona normalmente, mas na que possui outra tipagem para esse método ocorre um erro.
     2. Essa é uma limitação do typescript que precisamos ver como corrigir.
  2. Avançando no uso de Generics:
     1. Para corrigir o erro acima utilizamos o generics, isto é, na classe mãe colocamos um <T> na frente da classe, bem como na declaração de tipagem dos métodos dela, desse modo, nas classes filhas poderemos determinar qual é o tipo que elas usaram nesses métodos sem ter imcompatibilidade com a classe mãe, correndo tudo perfeitamente:
        1. Código de view.ts:

export class View<T> {

  protected elemento: HTMLElement;

  constructor(seletor:string) {

    this.elemento = document.querySelector(seletor);

  }

  template(model: T): string {

    throw Error('A classe filha deve implementar o método template')

  }

  update(model: T): void {

    const template = this.template(model);

    this.elemento.innerHTML = template;

  }

}

* + - 1. Código de mensagem-view.ts:

import { View } from "./view.js";

export class MensagemView extends View<string> {

  template(model: string): string {

    return `

      <p class="alert alert-info">${model}</p>

    `

  }

}

* + - 1. Código de negociacoes-view.ts:

import { Negociacoes } from "../models/negociacoes.js";

import { View } from "./view.js";

export class NegociacoesView extends View<Negociacoes> {

  template(model: Negociacoes): string {

    return `

    <table class="table table-hover table-bordered">

      <thead>

          <tr>

              <th>DATA</th>

              <th>QUANTIDADE</th>

              <th>VALOR</th>

          </tr>

      </thead>

      <tbody>

        ${model.lista().map(negociacao => {

          return `

            <tr>

                <td>${new Intl.DateTimeFormat().format(negociacao.data)}</td>

                <td>${negociacao.quantidade}</td>

                <td>${negociacao.valor}</td>

            </tr>

          `

        }).join('')}

      </tbody>

    </table>

    `;

  }

}

* 1. Classe abstrata:
     1. Decidimos anteriormente que nossa classe mãe view lançaria um erro se as classes filhas não sobrescrevessem o método template() a partir de um throw Error(), mas o desenvolvedor só saberia que ele esqueceu de sobrescrever esse método em uma classe filha em runtime, ou seja, quebra totalmente o propósito do TS.
     2. Para corrigir isso colocamos o modificador abstract antes da classe mãe e do método que queremos dar erro se não tiver sido sobrescrevida, desse modo, não deixando nosso código nem mesmo compilar se não tiver sido sobrescrito nas classes filhas. Código da classe mãe view.ts:

export abstract class View<T> {

  protected elemento: HTMLElement;

  constructor(seletor:string) {

    this.elemento = document.querySelector(seletor);

  }

  abstract template(model: T): string;

  update(model: T): void {

    const template = this.template(model);

    this.elemento.innerHTML = template;

  }

}

* + 1. Removemos todo o bloco de código do método, deixando somente a tipagem e dizendo que ele é abstrato.
    2. Esses recursos de generics, esticamente tipado, abstrato são todas coisas que só conseguimos fazer no TS, jamais seria possível fazer no JS. Por isso que é tão vantajoso utilizar essa linguagem.
  1. O que aprendemos:
     1. Herança com TypeScript;
     2. Classes com tipo genérico;
     3. Classes abstratas;
     4. O modificador protected.