**Download do projeto**

Baixe o projeto do curso [aqui](https://s3.amazonaws.com/caelum-online-public/typescript/01-alurabank.zip). Ele possui algumas bibliotecas que precisaremos utilizar ao longo do curso.

**TypeScript**

O compilador do TypeScript precisa da plataforma **Node.js** instalada para poder funcionar, **pré-requisito fundamental para este curso**. O segundo pré-requisito é o Visual Studio Code (VSCode). Como o TypeScript é uma linguagem criada pela Microsoft e o VSCode é mantido pela mesma empresa como projeto open source, este editor possui uma das melhores integrações com a linguagem TypeScript sem que o programador tenha que realizar dezenas de configurações ou instalar plugins.

Vejamos a seguir como proceder para instalar os pré-requisitos.

**Versão do Node.js**

É necessário no mínimo o Node.js em sua versão 6.X ou versões superiores **pares**!. Não instale versões ímpares do Node.js, pois não são LTS!\*\*

**Caso você já tenha o Node instalado**

Se você já tem o Node instalado em sua máquina, verifique se ele está pelo menos na versão **6.X** através do comando node -v no seu terminal, pois precisamos de uma versão atualizada do mesmo para que não ocorra erros na sua máquina durante o projeto.

**Instalação Node.js no Linux (Ubuntu)**

No Ubuntu, através do terminal (permissão de administrador necessária) execute o comando abaixo:

sudo apt-get install -y nodejs

**ATENÇÃO:** em algumas distribuições Linux, pode haver um conflito de nomes quando o Node é instalado pelo apt-get. Neste caso específico, no lugar do binário ser node, ele passa a se chamar **nodejs**. Isso gera problemas, pois a instrução **npm start** não funcionará, pois ela procura o binário node e não nodejs. Para resolver no Ubuntu

sudo ln -s /usr/bin/nodejs /usr/bin/node

depois o comando npm start funcionará conforme esperado.

É uma pena haver essa discrepância, mas fica aqui essa dica!

**Instalação Node.js no Windows**

Baixe o instalador clicando no grande botão install, diretamente da página do [Node.js](https://nodejs.org/). Durante a instalação, você apenas clicará nos botões para continuar o assistente. Não troque a pasta padrão do Node.js durante a instalação, a não ser que você saiba exatamente o que está fazendo.

**Instalação Node.js no MAC**

O [homebrew](http://brew.sh/) é a maneira mais recomendada para instalar o Node.js em sua máquina, através do comando:

brew update

brew install node

Não usa homebrew? Sem problema, baixe o instalador clicando no grande botão install, diretamente da página do [Node.js](http://brew.sh/).

**Visual Studio Code (Gratuito)**

O Visual Studio Code é um editor gratuito e multiplataforma (Mac, Linux e Windows) que se integra muito bem com o TypeScript com zero esforço por parte do desenvolvedor. Você pode baixá-lo no endereço <https://code.visualstudio.com/download>.

Se você utiliza IDE's como Eclipse ou NetBeans, sugiro abandonar essas IDE's pois você encontrará problemas desde acentuação até mensagens de erros "falso positivo". Caso você insista no uso delas, faça primeiro o curso usando o editor indicado e depois utilize sua IDE. Isso evitará que você poste no fórum dúvidas a respeito desses editores que não aconselho utilizar na parte de front-end.

## Download do projeto

Baixe o projeto do curso [aqui](https://s3.amazonaws.com/caelum-online-public/typescript/01-alurabank.zip). Ele possui algumas bibliotecas que precisaremos utilizar ao longo do curso.

## TypeScript

O compilador do TypeScript precisa da plataforma **Node.js** instalada para poder funcionar, **pré-requisito fundamental para este curso**. O segundo pré-requisito é o Visual Studio Code (VSCode). Como o TypeScript é uma linguagem criada pela Microsoft e o VSCode é mantido pela mesma empresa como projeto open source, este editor possui uma das melhores integrações com a linguagem TypeScript sem que o programador tenha que realizar dezenas de configurações ou instalar plugins.

Vejamos a seguir como proceder para instalar os pré-requisitos.

## Versão do Node.js

É necessário no mínimo o Node.js em sua versão 6.X ou versões superiores **pares**!. Não instale versões ímpares do Node.js, pois não são LTS!\*\*

## Caso você já tenha o Node instalado

Se você já tem o Node instalado em sua máquina, verifique se ele está pelo menos na versão **6.X** através do comando node -v no seu terminal, pois precisamos de uma versão atualizada do mesmo para que não ocorra erros na sua máquina durante o projeto.

## Instalação Node.js no Linux (Ubuntu)

No Ubuntu, através do terminal (permissão de administrador necessária) execute o comando abaixo:

sudo apt-get install -y nodejs

**ATENÇÃO:** em algumas distribuições Linux, pode haver um conflito de nomes quando o Node é instalado pelo apt-get. Neste caso específico, no lugar do binário ser node, ele passa a se chamar **nodejs**. Isso gera problemas, pois a instrução **npm start** não funcionará, pois ela procura o binário node e não nodejs. Para resolver no Ubuntu

sudo ln -s /usr/bin/nodejs /usr/bin/node

depois o comando npm start funcionará conforme esperado.

É uma pena haver essa discrepância, mas fica aqui essa dica!

## Instalação Node.js no Windows

Baixe o instalador clicando no grande botão install, diretamente da página do [Node.js](https://nodejs.org/). Durante a instalação, você apenas clicará nos botões para continuar o assistente. Não troque a pasta padrão do Node.js durante a instalação, a não ser que você saiba exatamente o que está fazendo.

## Instalação Node.js no MAC

O [homebrew](http://brew.sh/) é a maneira mais recomendada para instalar o Node.js em sua máquina, através do comando:

brew update

brew install node

Não usa homebrew? Sem problema, baixe o instalador clicando no grande botão install, diretamente da página do [Node.js](http://brew.sh/).

## Visual Studio Code (Gratuito)

O Visual Studio Code é um editor gratuito e multiplataforma (Mac, Linux e Windows) que se integra muito bem com o TypeScript com zero esforço por parte do desenvolvedor. Você pode baixá-lo no endereço <https://code.visualstudio.com/download>.

Se você utiliza IDE's como Eclipse ou NetBeans, sugiro abandonar essas IDE's pois você encontrará problemas desde acentuação até mensagens de erros "falso positivo". Caso você insista no uso delas, faça primeiro o curso usando o editor indicado e depois utilize sua IDE. Isso evitará que você poste no fórum dúvidas a respeito desses editores que não aconselho utilizar na parte de front-end.

Todo projeto tem um ponto de partida, o nosso será criação de um modelo de negociação. Um modelo nada mais é do que uma abstração de algo do mundo real. Um exemplo de modelo é aquele criado por um analista de mercado, permitindo-o simular operações e até prever seu comportamento.

No domínio de uma bolsa de valores, uma negociação possui a seguinte especificação:

* Obrigatoriamente tem uma data, quantidade e valor.
* Seu volume é calculado multiplicando-se a quantidade negociada no dia pelo valor negociado.
* Não pode ser modificada depois de realizada.

Agora que já temos informações suficientes, podemos dar início a materialização da especificação do modelo de negociação em nosso código. No paradigma orientado à objetos, utilizamos classes para essa finalidade.

## A classe Negociação

Vamos criar o script que declarará nossa classe Negociacao na pasta app/js/models/Negociacao.js:

// app/js/models/Negociacao.js

class Negociacao {

constructor(data, quantidade, valor) {

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;

}

}

Criamos uma especificação de uma negociação utilizando ES2015. Mas será que essa classe é fiel à especificação de uma negociação? Vejamos.

Vamos importar o script em app/index.html para que ele seja carregada.

<!-- client/index.html -->

<!-- código anterior omitido -->

<script src="js/models/Negociacao.js"></script>

</body>

<html>

Agora, vamos criar o arquivo app/js/app.js que será o ponto de entrada da nossa aplicação. Por enquanto, vamos criar apenas instâncias de negociações para testar nossa aplicação:

Por fim, não podemos esquecer de importá-lo em app/index.html:

<!-- client/index.html -->

<!-- código anterior omitido -->

<script src="js/models/Negociacao.js"></script>

<script src="js/app.js"></script>

</body>

<html>

Vamos testar a primeira regra. Uma negociação obrigatoriamente deve ter uma data, quantidade e valor:

// app/js/app.js

let negociacao = new Negociacao(new Date(), 2, 100);

console.log(negociacao);

Perfeito, no Console vemos os dados da negociação preenchidos. Mas será que nossa especificação de Negociação segue fielmente das regras que vimos? É isso que veremos no próximo vídeo.

Vamos verificar primeiro o cálculo do volume. Vamos lá:

// app/s/app.ts

let negociacao = new Negociacao(new Date(), 1, 100);

console.log(negociacao.volume); // exibe 100, correto!

Como não faz sentido criarmos uma negociação sem as três informações, no ato de criação já passamos esses dados para seu constructor(). No entanto, a linguagem JavaScript não me proíbe de fazer assim:

// app/js/app.js

let negociacao = new Negociacao();

console.log(negociacao);

Outro ponto é que depois de criada, uma negociação não pode ser modificada. Vamos realizar outro teste:

// app/js/app.js

let negociacao = new Negociacao(new Date(), 1, 100);

console.log(negociacao);

negociacao.quantidade = 3;

console.log(negociacao.quantidade);

Excelente, a atribuição negociacao.quantidade é ignorada, porque quantidade é um getter. No entanto, nada nos impede de acessarmos as propriedade com \_:

// app/js/app.js

let negociacao = new Negociacao(new Date(), 1, 100);

console.log(negociacao);

negociacao.\_quantidade = 3;

console.log(negociacao.quantidade); // modificou

O uso do prefixo é apenas uma convenção para a linguagem JavaScript indicando que a propriedade não deve ser alterar fora da classe, ou seja, uma convenção para indicar que a propriedade é privada.

Por mais que a linguagem JavaScript tenha evoluído, ela não conseguir lidar com questões como essa sem termos que apelar para convenções ou aplicar técnicas que aumenta bastante a complexidade do nosso código. Chegou a hora de mostrar um pouquinho do que o TypeScript pode fazer por nós.

Para que possamos utilizar o TypeScript precisamos da plataforma Node.js instalada. Aliás, essa plataforma já era um requisito de infraestrutura apontado no exercício obrigatório do capítulo.

É através do gerenciador de pacotes do Node.js que instalamos o TypeScript, mas primeiro, precisamos criar o arquivo package.json que nada mais é do que uma "caderneta" na qual temos registrados todos os módulos da aplicação baixados pelo npm.

Através do seu terminal favorito, vamos acessar a pasta alurabank. Dentro dela, vamos executar o comando:

npm init

Podemos teclar ENTER para todas as perguntas. No final, teremos o arquivo alurabank/package.json:

{

"name": "alurabank",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

Agora que temos nosso arquivo criado, vamos solicitar ao npm que instale o TypeScript para nós:

npm install typescript@2.3.2 --save-dev

Dentro de instantes ele será instalado dentro da pasta alurabank/node\_modules. Esse passo não é suficiente, precisamos configurar o compilador. Aliás, muitas IDE's escondem esses detalhes do desenvolvedor, mas inevitavelmente cedo ou tarde ele terá que lidar com essas configurações para poder alterar o comportamento do compilador do TypeScript. Este curso, mesmo sendo introdutório, o deixará seguro com tudo o que acontece por debaixo dos panos.

O próximo passo será renomearmos a pasta alurabank/app/js para alurabank/app/ts, inclusive vamos mudar a extensão dos arquivos app.js e Negociacao.jsrespectivamente para app.ts e Negociacao.ts. Afinal, a extensão .ts é aquela de todo arquivo TypeScript.

## O arquivo tsconfig.json

Precisamos criar o arquivo alurabank/tsconfig.json que guardará as configurações do nosso compilador.

{

"compilerOptions": {

"target": "es6",

"outDir": "app/js"

},

"include": [

"app/ts/\*\*/\*"

]

}

Nele, indicamos em compilerOptions as configurações do compilador. No caso, indicamos que o resultado final da compilação será um código compatível com es6 e que eles ficarão dentro da pasta app/js. Por fim, em include, indicamos o local onde o compilador deve buscar seus arquivos.

Excelente, temos a configuração mínima para que nosso compilador funcione, mas como o executaremos? Uma boa prática é criarmos um script em nosso package.jsonque se encarregará de chamá-lo para nós através do terminal.

Vamos alterar alurabank/package.json e adicionarmos o script:

"compile": "tsc"

Nosso package.json ficará assim:

{

"name": "alurabank",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"compile": "tsc"

},

"author": "",

"license": "ISC",

"devDependencies": {

"typescript": "^2.3.2"

}

}

Feche e abra o VSCode para que ele possa levar em consideração as configurações que realizamos no compilador.

Agora, através do terminal, ainda dentro da pasta alurabank faremos:

npm run compile

Após a conclusão do comando, veremos uma série de mensagens de erro, inclusive no próprio visual studio code veremos as mesmas mensagens em todo lugar que estiver sublinhado de vermelho. Isso significa que houve algum problema de compilação do nosso código que precisamos resolver, ou seja, alguma sintaxe não compatível com TypeScript.

O código de Negociacao.ts não compila, porque apesar do TS ser um superset do ES2015, ele necessita uma declaração especial para propriedade de classes. Essas são declaradas no corpo da classe:

class Negociacao {

// declaração das propriedades de classe

\_data;

\_quantidade;

\_valor;

constructor(data, quantidade, valor) {

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;

}

}

Se rodarmos nosso compilador mais uma vez através do comando npm run compilenenhuma mensagem de será exibida, indicando que nosso código compila. Um ponto curioso é que o resultado da compilação em app/js/models/Negociacao.js não possui as propriedades de classe. Está corretíssimo, pois se tivesse, não seria reconhecido pelo navegador. Lembre-se que todo código TypeScript deve ser transformado em um código em JavaScript para que seja compreendido pelo navegador.

Mas parece que escrevemos mais para pouca coisa, pois ainda podemos acessar a propriedade \_quantidade mesmo com um código escrito em TypeScript.

Podemos lançar mão de um recurso exclusivo da linguagem, o modificador privatepara tornar as propriedades da classe privadas. Propriedades privadas só podem ser acessadas pelos métodos da própria classe, resultando em um erro de compilação caso seja feito um acesso externo.:

// app/js/models/Negociacao.js

class Negociacao {

private \_data;

private \_quantidade;

private \_valor;

constructor(data, quantidade, valor) {

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;

}

}

Agora, se em app.ts tentarmos acessar a propriedade \_quantidade, por exemplo, receberemos um erro de compilação:

const negociacao = new Negociacao(new Date(), 1, 100);

negociacao.\_quantidade = 3; // O VSCODE indica um erro de compilação aqui

console.log(negociacao.quantidade);

Inclusive o VCode indicará para nós que estamos tentando acessar uma propriedade com modificador de acesso privado. Contudo, mesmo com o erro de compilação, será gerado um arquivo .js. Não queremos isso, queremos apenas arquivos .js gerados a partir de arquivos .ts sintaticamente corretos. Para isso, vamos adicionar em tsconfig.js mais uma configuração, a "noEmiteOnError":

{

"compilerOptions": {

"target": "es6",

"outDir": "app/js",

"noEmitOnError": true

},

"include": [

"app/ts/\*\*/\*"

]

}

Agora, só serão gerados arquivos .js apenas se o arquivo .ts não tiver nenhum erro de compilação.

Podemos automatizar o processo de compilação que será disparado toda vez que um arquivo .ts for modificado. Para isso, vamos adicionar mais um script em alurabank/package.json, o script "start": "tsc -w":

{

"name": "alurabank",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"compile": "tsc",

"start": "tsc -w"

},

"author": "",

"license": "ISC",

"devDependencies": {

"typescript": "^2.3.2"

}

}

Agora, no terminal e dentro da pasta aluraback, vamos executar o comando:

npm start

O terminal ficará travado pois o serviço de monitoramento de arquivos do TypeScript terá entrado em ação. Experimente alterar os arquivos .ts e veja o resultado. Só não esqueça de recarregar seu navegador para que as mudanças dos últimos arquivos .jsgerados sejam carregadas.

De nada nos adianta a classe Negociacao se o usuário não pode interagir com ela. Vamos criar uma classe que lidará com as ações do usuário, a classe NegociacaoController. Ela ficará no namespace controllers.

// app/ts/controllers/NegociacaoController.ts

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(event) {

event.preventDefault();

const negociacao = new Negociacao(

this.\_inputData.value,

this.\_inputQuantidade.value,

this.\_inputValor.value);

console.log(negociacao);

}

}

Nossa classe declara três propriedades que guardarão os elementos do DOM de entrada, no caso, os de data, quantidade e valor. No constructor() usamos document.querySelector para buscar os elementos. Por fim, no método adiciona, criamos uma nova instância de Negociacao com base nos dados do formulário.

Não podemos nos esquecer de importar o script gerado pelo processo de compilação:

<script src="js/models/Negociacao.js"></script>

<script src="js/app.js"></script>

<script src="js/controllers/NegociacaoController.js"></script>

Agora, vamos alterar app.ts, para que crie uma instância de NegociacaoControllerpara em seguida associar ao evento submit do formulário de index.html a chamada do método adiciona.

// app/ts/app.ts

const controller = new NegociacaoController();

document

.querySelector('.form')

.addEventListener('submit',controller.adiciona.bind(controller));

Nosso código compila, mas a ordem de carregamento de scripts nos causou problemas. O arquivo NegociacaoController.js deve ser carregado antes de app.ts e depois NegociacaoController, pois este depende de Negociacao. O sistema de namespace não ataca o problema da ordem de dependência entre scripts, algo que ainda resolveremos neste treinamento.

<!-- app/index.html -->

<!-- código anterior omitido -->

<script src="js/models/Negociacao.js"></script>

<script src="js/controllers/NegociacaoController.js"></script>

<script src="js/app.js"></script>

<!-- código posterior omitido -->

Isso é suficiente para fazer funcionar nossa aplicação.

Nossa instância de Negociacao esta recebendo como valores das propriedades \_data, \_quantidade e \_valor valor em string. Lembre-se que isso acontece porque os elementos do DOM que guardam os valores digitados pelo usuário sempre os guardam como string.

Como o nome da linguagem já diz, TypeScript é baseado em tipos. Mas se precisamos tipar nossas propriedades, como nosso código funcionou sem definirmos um tipo. Até porque, linguagens como Java ou C# não deixariam passar declarações sem a informação do tipo.

Isso acontece porque o padrão do TypeScript é adotar o tipo any implícito. O tipo anyindica que a variável, propriedade, parâmetro de método ou seu retorno podem ser de qualquer tipo. Esse comportamento ajuda na migração de um código legado em JavaScript para esta linguagem. No entanto, como estamos começando um projeto do zero, podemos impedir que o TypeScript assuma implicitamente o tipo any, forçando-nos a explicitar a tipagem.

Vamos alterar alurabak/tsconfig.json para adicionarmos no em seguida a configuração "noImplicitAny": true:

{

"compilerOptions": {

"target": "es6",

"outDir": "app/js",

"noEmitOnError": true,

"noImplicitAny": true

},

"include": [

"app/ts/\*\*/\*"

]

}

Feche e abra novamente o VCode para que ele entenda a nova configuração. Agora, nosso arquivo Negociacao.ts e NegociacaoController.ts apresentarão erros de compilação, pois não tipamos as propriedades da classe. Vamos começar por Negociacao.ts.

Podemos indicar o tipo das propriedades de uma classe através de : Tipo. Precisamos também tipar os parâmetros recebidos no constructor():

// app/ts/models/Negociacao.ts

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;

}

// código posterior omitido

Pode ser um tanto tedioso declarar das propriedades e também os parâmetros no constructor. TypeScript possui o seguinte atalho:

// app/ts/models/Negociacao.ts

class Negociacao {

constructor(

private \_data: Date,

private \_quantidade: number,

private \_valor: number) {}

No exemplo acima, os parâmetros do constructor() recebem o modificador private. Isso é suficiente para que o compilador do TypeScript receba os parâmetros considerando-os propriedades da classe.

Agora, precisamos resolver o problema de compilação em NegociacaoController.ts.

Durante nossas aulas, é provável que vocês tenha reparado que o Visual Studio Code, durante o processo de autocomplete, nos permite escolhermos além dos tipos string e number, os tipos String e Number. Há diferença?

Os tipos que começam em minúsculo equivalem a declaração literal. Vejamos um exemplo:

let nome = 'Flávio';

let idade = 20;

O TypeScript infere o tipo, sendo assim, a sintaxe é a mesma coisa que:

let nome: string = 'Flávio';

let idade: number = 20;

Se fizermos typeof nas duas variáveis temos como resultado string e numberrespectivamente:

let nome: string = 'Flávio';

console.log(typeof(nome)); // string

let idade: number = 20;

console.log(typeof(idade));// number

Contudo, JavaScript permite criar strings e números não como literais, mas como objetos:

let nome = new String('Flávio');

console.log(typeof(nome)); // Object

let idade = new Number(20);

console.log(typeof(idade)); // Object

Qual a diferença?

Os tipos string e number são literais e guardam um valor primitivo. Contudo, se tentarmos chamar algum método em variáveis declaradas com esses tipos, eles são empacotados automaticamente (auto-boxing) para String e Number respectivamente.

É por isso que esse código funciona:

let nome = 'Flávio';

nome.replace('/vio/', 'vião'); // faz auto-boxing

Dessa forma, TypeScript permite distinguir entre o tipo literal e o tipo objeto. Contudo, a boa prática é usarmos os tipos literais number e string, porque em JavaScript new String() e new Number() são raramente usados.

A classe NegociacaoController não compila. Precisamos ajustá-la. Mas qual será o tipo das propriedades da nossa classe? Passando o mouse em cima das instruções document.querySelector faz com que o VCode exiba a assinatura do método. Nele, identificamos que o retorno é do tipo Element. Perfeito, vamos utilizar este tipo nas propriedades da classe:

// app/ts/controllers/NegociacaoController.ts

class NegociacaoController {

private \_inputData: Element;

private \_inputQuantidade: Element;

private \_inputValor: Elementt;

// código posterior omitido

Excelente, agora precisamos tipar o parâmetro do método adiciona. Ele será do tipo Event. Inclusive, usando esse tipo, o VCode integrado com o TypeScript fará o autocomplete de propriedades e métodos para nós, inclusive se chamarmos um método que não existe, ele indicará o problema.

Agora, para nosso código compilar, precisamos tipar a parâmetro event do método adiciona(). No caso, utilizaremos o tipo Event:

// app/ts/controllers/NegociacaoController.ts

// código anterior omitido

adiciona(event: Event) {

// código omitido

}

Tudo parece estar no lugar, mas quando usamos o tipo Element nas propriedades da classe outro erro de compilação ocorreu em nosso código, dentro do método adiciona(). Agora, o TypeScript indica que as propriedades value não existem para elementos do tipo Element. E agora?

A classe NegociacaoController não compila. Precisamos ajustá-la:

// app/ts/controllers/NegociacaoController.ts

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

// código posterior omitido

Nossos inputs do formulário são do tipo HTMLInputElement. Inclusive, o próprio VSCode já nos lista todos os tipos possíveis da interface DOM. Por enquanto, entenda que o TypeScript já vem com uma lista de definições para esses tipos que são lidas automaticamente pelo VCode.

No entanto, nosso código ainda não compila, inclusive o compilador indica um erro agora dentro do constructor() de NegociacaoController. O problema é que document.querySelector() retorna elementos do tipo Element, um tipo mais genérico, pois seu retorno pode ser elementos que representam inputs, tabelas, destaques entre outros. Mas ao tratá-los todos como Element, deixamos de ter acesso a métodos e propriedades específicas dos elementos retornados, no caso, a propriedade value que todo input possui.

O primeiro passo para resolvermos nosso problema, é utilizarmos o tipo correto nas propriedades da nossa classe. Vamos utilizar o tipo HTMLInputElement:

// app/ts/controllers/NegociacaoController.ts

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

constructor() {

// continua com erro de compilação

this.\_inputData = document.querySelector('#data');

this.\_inputQuantidade = document.querySelector('#quantidade');

this.\_inputValor = document.querySelector('#valor');

}

O problema é que geramos um erro de compilação dentro do constructor() da classe. Não podemos pegar o valor de um tipo mais genérico e atribuí-lo a um tipo mais específico, apenas o contrário é permitido automaticamente pela linguagens. Porém, como temos certeza que estamos buscando elementos do tipo input, podemos assumir a responsabilidade de conversão, isto é, realizando uma conversão explícita:

// app/ts/controllers/NegociacaoController.ts

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

constructor() {

this.\_inputData = <HTMLInputElement>document.querySelector('#data');

this.\_inputQuantidade = <HTMLInputElement>document.querySelector('#quantidade');

this.\_inputValor = <HTMLInputElement>document.querySelector('#valor');

}

Esse processo de conversão explícita é chamado de casting. No caso, estamos forçando uma conversão de um tipo mais genérico para um tipo mais específico. Pode ocorrer o casting implícito, quando atribuímos um tipo mais específico a um tipo mais genérico. Por enquanto, apenas o casting explícito se apresentou necessário em nossa aplicação.

Estamos quase lá! Mas por mais que tenhamos resolvido os problemas de compilação, novos apareceram. No método adiciona, como o retorno de um HTMLInputElement é sempre uma string, ela não se coaduna com os tipos esperados pelo constructor() de Negociacao. Precisamos converter os dados antes que sejam passados para o constructor().

Mas ainda há um erro de compilação no método adiciona. Agora, como o TypeScript sabe que o valor retornado por um HTMLInputElement é sempre uma string, a passagem dos valores para o constructor de Negociacao não estão mais compatíveis.

// erro de compilação no método !

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

this.\_inputData.value,

this.\_inputQuantidade.value,

this.\_inputValor.value);

console.log(negociacao);

}

[ts] Argument of type 'string' is not assignable to parameter of type 'Date'..

Precisamos converter as string para o respectivo tipo esperado pelo constructor() de Negociacao. No final, nossa classe NegociacaoController estará assim:

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

constructor() {

this.\_inputData = <HTMLInputElement>document.querySelector('#data');

this.\_inputQuantidade = <HTMLInputElement>document.querySelector('#quantidade');

this.\_inputValor = <HTMLInputElement>document.querySelector('#valor');

}

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value));

console.log(negociacao);

}

}

O <input type="date"> não é suportado pelo Firefox por padrão, sendo necessário habilitar "No Firefox digite na barra de endereços about:config, pesquise da chave dom.forms.datetime e altere seu valor para true". No curso de JavaScript avançado aprendemos a lidar com datas independente do navegador no qual estamos trabalhando e isso também se aplica para um código em TypeScript. Sendo assim, é importante que você use o Chrome até o final do treinamento, pois nosso foco é TypeScript.

Contudo, se você usa o Microsoft Edge, apesar dele suportar este tipo de input, seu comportamento não sai como esperado ao criarmos uma data com a estratégia de this.\_inputData.value.replace(/-/g, '/')). No caso, podemos trocar de hífen para uma /:

adiciona(event: Event) {

event.preventDefault();

// mudou aqui

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, '/')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value));

console.log(negociacao);

}

Essa mudança de comportamento funciona tanto no Chrome quanto no Edge (pelo menos nas versões localizadas que testamos). Contudo, ainda é extremamente recomendado que você utilize o Chrome até o final do treinamento.

Vamos continuar com nosso projeto. Já somos capazes de criar uma instância de Negociacao a partir dos dados do formulário. Agora, precisamos guardar em uma lista todas as negociações incluídas pelo usuário para mais tarde exibirmos todas elas através de uma tabela.

No entanto, só podemos adicionar negociações em nossa lista, não podemos remover ou apagá-la. Um array do JavaScript permite dezenas de operações, sendo assim, vamos encapsular um array dentro da classe Negociacoes e fazer com que todo acesso ao array encapsulado passe pelos métodos da classe.

Vamos criar o arquivo app/ts/models/Negociacoes.ts. Ele terá uma única propriedade, um array:

class Negociacoes {

// erro de compilação, não podemos deixar o tipo any por padrão

// Member 'negociacoes' implicitly has an 'any[]' type.

private \_negociacoes = [];

}

Parece que desagradamos o TypeScript na declaração do nosso array. Lembre-se que agora precisamos identificar os tipos das propriedades da classe. A boa notícia é que o TypeScript já possui o tipo Array:

class Negociacoes {

// ainda continua com erro!

private negociacoes: Array = [];

}

Pois é, nosso compilador ainda não satisfeito. Mesmo identificando o tipo Array ainda temos a seguinte mensagem de erro:

[ts] Generic type 'Array<T>' requires 1 type argument(s).

O problema é que Array é um tipo genérico. Mas por que genérico? Podemos guardar strings, números e objetos dentro de um array e utilizarmos apenas Array indica que temos um array, mas não o tipo dos dados que contém. Dessa maneira, precisamos especificar o tipo de dados armazenados por um array da seguinte maneira:

class Negociacoes {

private \_negociacoes: Array<Negociacao> = [];

}

Veja que a declaração do array segue o padrão Array<TipoDosDadosQueContem>. Agora, o compilador do TypeScript ficará feliz.

Inclusive, podemos declarar um array do tipo Negociacao de uma maneira alternativa, pasta usarmos o tipo seguido de []:

class Negociacoes {

private \_negociacoes: Negociacao[] = [];

}

Agora, vamos para o método adiciona(), que deve receber como parâmetro um objeto do tipo Negociacao

class Negociacoes {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao) {

this.\_negociacoes.push(negociacao);

}

}

Qualquer outro tipo será vetado pelo compilador do TypeScript. Por fim, vamos criar o método paraArray que devolve o array encapsulado por Negociacoes:

class Negociacoes {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao) {

this.\_negociacoes.push(negociacao);

}

paraArray() {

return this.\_negociacoes;

}

}

Nosso código compila, deixamos o TypeScript feliz!

Para não corremos o risco de esquecermos, vamos importar o javascript que será resultante da compilação da classe em index.html. Vamos importá-lo antes de app.ts. Aliás, deixaremos sempre app.ts como último.

<!-- app/index.html -->

<!-- código anterior omitido -->

<script src="js/models/Negociacao.js"></script>

<script src="js/controllers/NegociacaoController.js"></script>

<script src="js/models/Negociacoes.js"></script>

<script src="js/app.js"></script>

Agora que já temos nosso novo modelo, já podemos integrá-lo com NegociacaoController.

Agora que temos nosso modelo de Negociacoes, vamos utilizá-lo em NegociacaoController. Primeiro, vamos criar a propriedade \_negociacoes na classe:

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

private \_negociacoes: Negociacoes = new Negociacoes();

Além de definirmos o tipo, já inicializamos a variável com uma instância de Negociacoes. Contudo, podemos remover a declaração do tipo que nosso código continuará a compilar.

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

// removeu o tipo!

private \_negociacoes = new Negociacoes();

Mas como isso é possível? O Compilador do TypeScript é inteligente o suficiente para entender que, se estamos criando uma instância de Negociacoes, a propriedade não tipada que receber seu valor assumirá o tipo Negociacoes.

Agora, vamos alterar o método adiciona() de NegociacaoController e armazenar cada negociação criada no modelo Negociacoes:

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value)

);

this.\_negociacoes.adiciona(negociacao);

// imprime a lista de negociações encapsulada

console.log(this.\_negociacoes.paraArray());

}

Excelente, a cada inclusão, conseguimos ver que a lista vai aumentando de tamanho. Mas que tal, ao invés de imprimirmos a referência do Array, imprimirmos cada elemento individualmente para visualizar mais facilmente os dados das negociações salvas?

class NegociacaoController {

// código anterior omitido

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value)

);

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoes.paraArray().forEach(negociacao => {

console.log(negociacao.data);

console.log(negociacao.quantidade);

console.log(negociacao.valor);

});

}

}

Nossa aplicação compila e funciona. Inclusive, o TypeScript conseguiu inferir que o tipo retornado por this.\_negociacao.paraArray() é do tipo Negociacao[]. Aliás, essa inferência é baseado no tipo do objeto retornado. Com isso, podemos usar o autocomplete e, se tentarmos chamar algum método ou propriedade que não exista, somos avisados.

No Entanto, a definição da nossa classe Negociacoes possui um problema. Lembre-se que só podemos realizar operações de inclusão no array encapsulado por Negociacoes. Aliás, este foi o motivo de termos criado a classe. Mas conseguimos acessar o array encapsulado por Negociacoes através do método paraArray(). Como temos acesso ao array desprotegido, podemos apagá-lo. Vejamos:

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value)

);

this.\_negociacoes.adiciona(negociacao);

// apaga o array

this.\_negociacoes.paraArray().length = 0; // acabou de apagar!

// não tem dado para iterar!

this.\_negociacoes.paraArray().forEach(negociacao => {

console.log(negociacao.data);

console.log(negociacao.quantidade);

console.log(negociacao.valor);

});

}

Temos o encapsulamento do array de negociações quebrado. Podemos resolver isso facilmente através da programação defensiva, retornando um novo array toda vez que o método paraArray() for chamado, sendo assim, qualquer mudança será efetuada na cópia e não no array original encapsulado por Negociacoes:

class Negociacoes {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao) {

this.\_negociacoes.push(negociacao);

}

paraArray() {

return [].concat(this.\_negociacoes);

}

}

Perfeito! Não conseguimos mais alterar o array encapsulado por Negociacoes. No entanto, algo aconteceu em nosso código. Se tentarmos usar o autocomplete dos métodos das instâncias de Negociacao do array retornado por paraArray() nada acontece. O motivo disso é que ao retornarmos um novo array [] concatenado com os itens do array encapsulado, o TypeScript passa a entender o retorno do método paraArray como sendo do tipo any[].

Você devem estar se perguntando. Mas nós configuramos o compilador para não aceitar o tipo any, certo? Perfeito, mas no caso de retornos de métodos o TypeScript não realiza essa restrição. E agora?

Podemos voltar a usufruir do autocomplete tipando o retorno do método paraArray, dessa forma, o TypeScript não tentará inferir o tipo do retorno e considerará o tipo especificado.

Alterando nosso código:

class Negociacoes {

private \_negociacoes: Negociacao[] = [];

adiciona(negociacao: Negociacao) {

this.\_negociacoes.push(negociacao);

}

paraArray(): Negociacao[] {

return [].concat(this.\_negociacoes);

}

}

Tipamos métodos adicionando :Tipo imediatamente os (). Agora, o TypeScript voltou a considerar o retorno de paraArray como sendo do tipo Negociacao[] e com isso podemos usar toda a checagem disponibilizada pela linguagem com interagirmos com a lista. Aliás, é uma boa prática expliciar o retorno de métodos e funções. Quando fazemos isso, se cometermos alguma gafe em nosso código no bloco do método ou da função retornando um tipo não esperado, seremos alertados pelo compilador. A partir de agora adotaremos esta postura.

Nossa classe NegociacaoController no final estará assim, removendo os testes que fizemos no método adiciona():

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

private \_negociacoes = new Negociacoes();

constructor() {

this.\_inputData = <HTMLInputElement>document.querySelector('#data');

this.\_inputQuantidade = <HTMLInputElement>document.querySelector('#quantidade');

this.\_inputValor = <HTMLInputElement>document.querySelector('#valor');

}

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value)

);

this.\_negociacoes.adiciona(negociacao);

}

}

Agora que já temos uma lista de negociações, que tal exibirmos essa lista para o usuário? Assunto do próximo vídeo!

De nada adianta termos uma lista de negociações se não somos capazes de exibí-la para o usuário. Precisamos construir dinamicamente uma tabela que represente os dados mais atualizados. Temos duas formas de fazer isso. A primeira, imperativa, é realizarmos o passo a passo de cada mutação que desejamos realizar no DOM, já a segunda, declarativa, podemos utilizar um template que pode ser compilado resultado em todas as mutações necessárias que precisamos realizar. Seguiremos a segunda forma.

Vamos declarar nosso template no mundo JavaScript. Se você já trabalhou com React do Facebook, ele também declara templates no mundo JavaScript com JSX. Utilizaremos uma solução que envolve apenas JavaScript.

Vamos criar o arquivo classe app/ts/views/NegociacoesView.ts. Nele, temos a classe NegociacoesView que possuirá apenas o método template(), que retornará uma string com a estrutura da tabela que desejamos construir:

// app/ts/views/NegociacoesView.ts

class NegociacoesView {

template(): string {

return `

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

<thead>

<tr>

<th>DATA</th>

<th>QUANTIDADE</th>

<th>VALOR</th>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

</tbody>

<tfoot>

</tfoot>

</table>

`

}

}

Utilizamos template string pela comunidade de quebras de linha, além de permitir interpolações sem o uso do operador +.

Antes de continuarmos, vamos importar o JavaScript resultante da compilação do nosso arquivo.

<!-- app/index.html -->

<!-- código anterior omitido -->

<script src="js/models/Negociacao.js"></script>

<script src="js/controllers/NegociacaoController.js"></script>

<script src="js/models/Negociacoes.js"></script>

<script src="js/views/NegociacoesView.js"></script>

<script src="js/app.js"></script>

<!-- código posterior omitido -->

Recarregando nossa página, vamos realizar um teste no Console do navegador:

// NO CONSOLE DO NAVEGADOR

view = new NegociacoesView();

view.template(); // exibe o template retornado

Excelente, mas agora precisamos indicar onde em nosso app/index.html nosso template será renderizado. Faremos isso adicionando uma <div> com o ID negociacoesView:

<div id="negociacoesView"></div>

<script src="js/models/Negociacao.js"></script>

<script src="js/controllers/NegociacaoController.js"></script>

<script src="js/models/Negociacoes.js"></script>

<script src="js/views/NegociacoesView.js"></script>

<script src="js/app.js"></script>

Agora, vamos criar a propriedade \_negociacoesView em NegociacaoController. Ela guardará uma instância de NegociacoesView:

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

private \_negociacoes = new Negociacoes();

// vai dar um erro de compilação, pois a classe não recebe parâmetro ainda

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

Estamos passando para o constructor de NegociacoesView um seletor CSS que indica o elemento do DOM que receberá a compilação do nosso template. No entanto, a classe ainda não esta preparada para receber esse parâmetro.

A ideia é seguinte. Com base no seletor recebido, NegociacoesView guardará internamente uma referência para o elemento do DOM referente ao seletor. Como teremos uma referência para o elemento, podemos atualizá-la com o resultado do template. O tipo da propriedade será Element, um tipo bem genérico, aceitando receber tipos mais especializados do DOM. Ele é suficiente porque disponibiliza o setter innerHTML que ao receber uma string, internamente a converte para elementos do DOM.

class NegociacoesView {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

template(): string {

return `

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

<thead>

<tr>

<th>DATA</th>

<th>QUANTIDADE</th>

<th>VALOR</th>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

</tbody>

<tfoot>

</tfoot>

</table>

`

}

}

O método template() é aquele no qual definimos o template da view, já o novo método update() que adicionaremos é aquele que ao ser chamado, atribuirá o resultado de template() à propriedade innerHTML do elemento do DOM:

class NegociacoesView {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(): void {

this.\_elemento.innerHTML = this.template();

}

template(): string {

return `

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

<thead>

<tr>

<th>DATA</th>

<th>QUANTIDADE</th>

<th>VALOR</th>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

</tbody>

<tfoot>

</tfoot>

</table>

`

}

}

Já podemos testar nosso código no terminal.

// NO CONSOLE DO NAVEGADOR

view = new NegociacoesView('#negociacoesView');

view.update(); // atualiza o elemento do DOM com os dados do template

A questão agora, é que o método update() precisa receber o modelo, ou seja, o dado no qual a função template() se baseará para gerar o template.

class NegociacoesView {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: Negociacoes): void {

this.\_elemento.innerHTML = this.template(model);

}

template(modelo: Negociacoes): string {

return `

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

<thead>

<tr>

<th>DATA</th>

<th>QUANTIDADE</th>

<th>VALOR</th>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

</tbody>

<tfoot>

</tfoot>

</table>

`

}

}

Agora, precisamos gerar uma tr para cada negociação encapsulada por Negociacoes:

class NegociacoesView {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: Negociacoes) {

this.\_elemento.innerHTML = this.template(model);

}

template(model: Negociacoes): string {

return `

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

<thead>

<tr>

<th>DATA</th>

<th>QUANTIDADE</th>

<th>VALOR</th>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

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

`

<tr>

<td>${negociacao.data.getDate()}/${negociacao.data.getMonth()+1}/${negociacao.data.getFullYear()}</td>

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

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

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

</tr>

`).join('')}

</tbody>

<tfoot>

</tfoot>

</table>

`

}

}

Por fim, vamos chamar o método this.\_negociacoesView.update(this.\_negociacoes) na inicialização do nosso controller e após a inclusão de novas negociações.

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

private \_negociacoes = new Negociacoes();

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

constructor() {

this.\_inputData = <HTMLInputElement>document.querySelector('#data');

this.\_inputQuantidade = <HTMLInputElement>document.querySelector('#quantidade');

this.\_inputValor = <HTMLInputElement>document.querySelector('#valor');

// atualiza a view para exibir os dados do modelo, vazio

this.\_negociacoesView.update(this.\_negociacoes);

}

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value)

);

this.\_negociacoes.adiciona(negociacao);

// depois de adicionar, atualiza a view novamente para refletir os dados

this.\_negociacoesView.update(this.\_negociacoes);

}

}

Podemos adicionar quantas negociações desejamos que todas serão exibidas em nossa tabela.

Nossa aplicação consegue incluir e exibir negociações. Mas é bem comum uma aplicação informar o usuário sobre o sucesso das operações realizadas. Para isso, vamos criar o arquivo app/ts/views/MensagemView.ts. Ele terá uma estrutura praticamente idêntica ao arquivo de view que criamos antes, a diferença estará apenas no seu template, que recebe um parâmetro diferente e retorna um template específico:

class MensagemView {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: string) {

this.\_elemento.innerHTML = this.template(model);

}

template(model: string): string {

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

}

}

Como todos os arquivo que criamos, vamos importar o resultado da sua compilação em app/index.html:

<!-- app/index.html -->

<!-- código anterior omitido -->

<script src="js/models/Negociacao.js"></script>

<script src="js/controllers/NegociacaoController.js"></script>

<script src="js/models/Negociacoes.js"></script>

<script src="js/views/NegociacoesView.js"></script>

<script src="js/views/MensagemView.js"></script>

<script src="js/app.js"></script>

<!-- código posterior omitido -->

Agora, vamos seguir um processo parecido como os que fizemos com a NegociacoesView, a diferença é que não chamaremos a função update() na inicialização, apenas no método adiciona():

class NegociacaoController {

private \_inputData: HTMLInputElement;

private \_inputQuantidade: HTMLInputElement;

private \_inputValor: HTMLInputElement;

private \_negociacoes = new Negociacoes();

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

private \_mensagemView = new MensagemView('#mensagemView');

constructor() {

this.\_inputData = <HTMLInputElement>document.querySelector('#data');

this.\_inputQuantidade = <HTMLInputElement>document.querySelector('#quantidade');

this.\_inputValor = <HTMLInputElement>document.querySelector('#valor');

this.\_negociacoesView.update(this.\_negociacoes);

}

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.value.replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.value),

parseFloat(this.\_inputValor.value)

);

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso');

}

}

A cada negociação adicionada, a mensagem é exibida. Tudo funciona, mais há algum "cheiro" de código que paira no ar. Assunto do próximo vídeo.

Se compararmos os arquivos NegociacoesView e MensagemView lado a lado, veremos que ambos são praticamente parecidos. Por exemplo, ambos possuem a propriedade \_elemento e um constructor() que recebem uma string.

Vamos isolar esse código repetido dentro da classe View, que declararemos no arquivo app/ts/views/View.ts:

class View {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

}

Agora, vamos remover a propriedade \_elemento e o constructor() de MensagemView e NegociacaoView. Com certeza ficaremos com erro de compilação.

Agora, utilizando um conceito do paradigma orientado a objetos, vamos fazer com que NegociacaoView e MensagemView herdem o código que centralizamos em View. Fazemos isso através da instrução extends.

class NegociacoesView extends View{

update(model: Negociacoes) {

// erro de compilação

this.\_elemento.innerHTML = this.template(model);

}

// código posterior omitido

class MensagemView extends View {

update(model: string) {

// erro de compilação

this.\_elemento.innerHTML = this.template(model);

}

// código posterior omitido

Quando usamos a instrução extends, podemos fazer com que uma classe herde de outra suas propriedades e métodos, inclusive constructor(), no entanto nosso código não compila. Recebemos a mesma mensagem de erro para NegociacoesView e MensagemView:

Property '\_elemento' is private and only accessible within class 'View'.

O problema é que propriedades declaradas com o modificador private não podem ser acessadas nem mesmo pelas classes filhas. Podemos resolver esse erro de compilação relaxando um pouco o nível de acesso à propriedade, utilizando o modificador protected em vez de private:

class View {

// mudou para protected!

protected \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

}

Agora, automaticamente, o VSCode deixará de exibir as mensagens de erro. Inclusive, podemos recarregar nossa página e testar nossa aplicação, que continuará funcionando.

Mas será que podemos reutilizar mais código?

Se olharmos atentamente nossas classes de view, vemos que o método update e template são bem parecidos. No entanto, o método template() é bem específico da classe filha, pois retorna strings especializadas. Que tal movermos o método update()para dentro de View?

class View {

protected \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: string) {

// erro de compilação

this.\_elemento.innerHTML = this.template(model);

}

}

Ops! Temos um erro de compilação, pois o método update depende do método template(). Temos que mover o métodotemplate` também.

class View {

protected \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: string) {

this.\_elemento.innerHTML = this.template(model);

}

template(model: string): string {

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

}

}

No caso do método template, vamos deixá-lo vazio, lançando uma exceção forçando as classes filhas a implementá-lo:

class View {

protected \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: string) {

this.\_elemento.innerHTML = this.template(model);

}

template(model: string): string {

throw new Error('Você deve implementar o método template');

}

}

Agora, nas classes NegociacoesView e MensagemView vamos remover o método update(). A classe MensagemView compila, mas NegociacoesView não.

Class 'NegociacoesView' incorrectly extends base class 'View'.

Types of property 'template' are incompatible.

Type '(model: Negociacoes) => string' is not assignable to type '(model: string) => string'.

Types of parameters 'model' and 'model' are incompatible.

Type 'string' is not assignable to type 'Negociacoes'.

O problema é que em View temos fixos como parâmetro de update e template o tipo string, mas no caso de NegociacoesView o tipo precisa ser Negociacoes. E agora?

Podemos lançar mão de um recurso chamado **generics**. Vejamos a declaração da classe View com este recurso:

class View<T> {

protected \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: T) {

this.\_elemento.innerHTML = this.template(model);

}

template(model: T): string {

throw new Error('Você deve implementar o método template');

}

}

Vejam a sintaxe class View<T>. Nela, estamos indicando que a classe ao ser usada pode receber um tipo como indicação e esse tipo será usado em todos os lugares que Testiver presente.

Agora, quem for herdar de View, precisará indicar o tipo que desejamos utilizar na classe:

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>

<th>VOLUME</th>

</tr>

</thead>

<tbody>

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

`

<tr>

<td>${negociacao.data.getDate()}/${negociacao.data.getMonth()+1}/${negociacao.data.getFullYear()}</td>

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

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

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

</tr>

`).join('')}

</tbody>

<tfoot>

</tfoot>

</table>

`

}

}

class MensagemView extends View<string> {

template(model: string): string {

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

}

}

Nosso código continua compilando! Agora, não precisamos mais do modificador protected, pois agora acessarmos a propriedade através dos métodos da classe:

class View<T> {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: T) {

this.\_elemento.innerHTML = this.template(model);

}

template(model: T): string {

throw new Error('Você deve implementar o método template');

}

}

Mas será que podemos melhorar nosso código?

Para saber mais: preciso de protected mesmo?

Vejamos a classe View que criamos neste capítulo:

abstract class View<T> {

protected \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: T) {

this.\_elemento.innerHTML = this.template(model);

}

abstract template(model: T): string;

}

Usamos o modificador protected logo no início para permitir que os métodos updatedefinidos nas classes filhas pudessem acessar a propriedade. Contudo, melhoramos o design de nossa classe e conseguimos fazer com que as classes filhas também herdassem o método update de View. Nesse contexto, o acesso à propriedade \_elemento é feito por um método da classe pai, sendo assim, podemos voltar com o modificador private em nossa classe, evitando assim o relaxamento do seu encapsulamento:

abstract class View<T> {

// voltando com private!

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: T) {

this.\_elemento.innerHTML = this.template(model);

}

abstract template(model: T): string;

}

Era essa ideia que este instrutor gostaria ter passado para vocês quando disse no vídeo que haveria outra solução para acessar a propriedade \_elemento sem a necessidade do uso do modificador protected. Mas foi tanta emoção que deixei de fazer essa pequena alteração.

Do jeito que o nosso código está, o programador não é avisado em tempo de compilação caso esqueça de sobrescrever o método template. Só saberemos do erro em tempo de runtime. Não queremos essa solução, pois será muito tarde para detectar o problema.

Para resolvermos esse problema, podemos tornar View uma classe *abstrata\**. Classes abstratas não podem ser instanciadas, apenas herdadas. Essas classes também podem definir um ou mais métodos abstratos que nada mais são do que métodos sem implementação que precisam ser implementados pelas classes filhas.

Alterando nossa classe View:

abstract class View<T> {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = document.querySelector(seletor);

}

update(model: T) {

this.\_elemento.innerHTML = this.template(model);

}

abstract template(model: T): string;

}

Veja que tanto a classe quando o método template são abstratos. Se não implementarmos o método template nas classes filhas, o TypeScript indicará um erro, lembrando o programador de sua tarefa.

Nossa aplicação continua funcionando e evoluindo. Realizamos manipulação de DOM em alguns pontos da nossa aplicação. Apesar da nossa manipulação não gastar muitas linhas de código, podemos incorporar o jQuery em nosso projeto. Essa biblioteca de mais de 10 anos conquistou o coletivo imaginários dos desenvolvedores front-end e ainda é utilizada em vários projetos, possuindo um ecossistema de plugins gigantesco. Com ela, podemos nos blindar com problemas de compatibilidade de muitos smartphones Android que usam versões antigas do Webkit, que foi substituído pelo Blink nas novas versões do Chrome. Além disso, podemos utilizar a sintaxe elegante que o jQuery oferece.

Já temos o script do jQuery dentro da pasta alurabank/app/lib. Vamos importá-lo em app/index.html como primeiro script:

<!-- app/index.html -->

<!-- código anterior omitido -->

<script src="lib/jquery.min.js"></script>

<script src="js/models/Negociacao.js"></script>

<script src="js/controllers/NegociacaoController.js"></script>

<script src="js/models/Negociacoes.js"></script>

<script src="js/views/View.js"></script>

<script src="js/views/NegociacoesView.js"></script>

<script src="js/views/MensagemView.js"></script>

<script src="js/app.js"></script>

<!-- código anterior -->

Agora que já temos o jQuery importado em nosso projeto, vamos começar alterando a classe app/ts/views/View.ts. Em vez de utilizarmos document.querySelector, vamos utilizar o famoso aliás para o jQuery, o $:

abstract class View<T> {

private \_elemento: Element;

constructor(seletor: string) {

// erro de compilação

this.\_elemento = $(seletor);

}

update(model: T) {

this.\_elemento.innerHTML = this.template(model);

}

abstract template(model: T): string;

}

Em nossa primeira alteração, recebemos um erro de compilação:

Cannot find name '$'.

O jQuery vive no escopo global e através do JavaScript podemos acessá-lo sem delongas. Contudo, com TypeScript, o compilador não entende que $ vive no escopo global e espera encontrar a declaração da variável $ em nosso código. Faz todo sentido, pois o compilador pode nos avisar caso estejamos acessando uma variável que não foi declarada ou que escrevermos seu nome errado.

Podemos fazer com que o compilador faça "vista grossa" para a variável $. Para isso, no início do nosso arquivo, vamos colocar a instrução declare var $:any:

declare var $: any;

abstract class View<T> {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = $(seletor);

}

update(model: T) {

this.\_elemento.innerHTML = this.template(model);

}

abstract template(model: T): string;

}

Com essa manobra, silenciamos o compilador. Agora, para ele, $ é um objeto do tipo any. Por fim, no método update, vamos trocar de this.\_elemento.innerHTML para this.\_elemento.html, o método do jQuery Object responsável em atualizar o elemento do DOM com base numa string:

declare var $: any;

abstract class View<T> {

private \_elemento: Element;

constructor(seletor: string) {

this.\_elemento = $(seletor);

}

update(model: T) {

// erro de compilação

// Property 'html' does not exist on type 'Element'

this.\_elemento.html(this.template(model));

}

abstract template(model: T): string;

}

Ganhamos outro erro de compilação. Faz sentido, porque o tipo Element não possui o método html. E agora? Vamos mudar a propriedade \_elemento do tipo Element para any. Isso é possível, pois estamos definindo o tipo any explicitamente:

declare var $: any;

abstract class View<T> {

protected \_elemento: any;

constructor(seletor: string) {

this.\_elemento = $(seletor);

}

update(model: T) {

this.\_elemento.html(this.template(model));

}

abstract template(model: T): string;

}

Perfeito, nosso código compila, inclusive podemos testar nossa aplicação e tudo funcionará, mas essa solução deixa muito à desejar. Primeiro, trabalhando com o tipo any não temos qualquer verificação do compilador do TypeScript e nem autocomplete. Estamos abdicando de um recurso poderoso da linguagem. Por exemplo, podemos escrever errado o nome do método htm e o TypeScript não saberá nos dizer antecipadamente que o método não existe. E agora?

A boa notícia é que o TypeScript possui um recurso interessantíssimo, que pode trazer de volta sua análise sintática, checagem de tipos e autocomplete não só com jQuery, mas com várias bibliotecas famosas do mercado.

jQuery foi criado com TypeScript? Com certeza não e, se não foi, não possui variáveis, propriedades e parâmetro de métodos e funções tipados. Então, como suportar os recursos da linguagem TypeScript com uma biblioteca que não foi criada utilizando nessa linguagem?

Os criadores da biblioteca ou terceiros podem criar um arquivo chamado TypeScript Declaration File. Este arquivo possui informações dos nomes de métodos e funções, inclusive tipos que podem ser utilizados pelo TypeScript. Quando carregado, o TypeScript conseguirá, baseado nesse arquivo, realizar checagem estática inclusive lançar mão de todos seu poder através de seu editor ou IDE favorita.

No caso, vamos instalar o tipo do jQuery. Vale lembrar que esse tipo não foi definido pela equipe do jQuery:

npm install @types/jquery@2.0.42 --save-dev

Você pode instalar o type definiton da sua biblioteca favorita, contanto que ela exista. Inclusive podem haver arquivos de definições criados por mais de um colaborador, no final, somos nós que devemos escolher o arquivo que for mais atualizado. Não há solução mágica, é necessário realizar pesquisas na Internet. Inclusive, pode ser que nem existe um arquivo tsd para sua biblioteca favorita, sendo assim, a solução com declare var continua sendo válida.

Vamos abrir e fechar o Visual Studio Code. Por padrão, o compilador do TypeScript procurará por padrão todos os @types dentro da pasta node\_modules sem termos que nos preocupar.

Agora, com o VSCode reaberto, vamos remover o declare var . Em seguida, vamos usar o tipo jQuery para \_elemento, propriedade da nossa classe. Esse tipo não está disponível antes, passou a estar depois de termos instalado o tds do jQuery:

abstract class View<T> {

private \_elemento: JQuery;

constructor(seletor: string) {

this.\_elemento = $(seletor);

}

update(model: T) {

this.\_elemento.html(this.template(model));

}

abstract template(model: T): string;

}

Fantástico! Agora, podemos usar o jQuery na declaração da nossa classe NegociacaoController:

class NegociacaoController {

private \_inputData: JQuery;

private \_inputQuantidade: JQuery;

private \_inputValor: JQuery;

private \_negociacoes = new Negociacoes();

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

private \_mensagemView = new MensagemView('#mensagemView');

constructor() {

this.\_inputData = $('#data');

this.\_inputQuantidade = $('#quantidade');

this.\_inputValor = $('#valor');

this.\_negociacoesView.update(this.\_negociacoes);

}

adiciona(event: Event) {

event.preventDefault();

const negociacao = new Negociacao(

new Date(this.\_inputData.val().replace(/-/g, ',')),

parseInt(this.\_inputQuantidade.val()),

parseFloat(this.\_inputValor.val())

);

this.\_negociacoes.adiciona(negociacao);

this.\_negociacoesView.update(this.\_negociacoes);

this.\_mensagemView.update('Negociação adicionada com sucesso');

}

}

// app/ts/app.ts

const controller = new NegociacaoController();

$('.form').submit(controller.adiciona.bind(controller));

Bibliotecas como Lodash, Underscore, inclusive plugins do jQuery UI possuem seu respectivo arquivo tsd. Um repositório do git que centraliza algumas desses arquivo é <https://github.com/DefinitelyTyped/DefinitelyTyped>.

No npm, existe uma série de TypeScript definitons files para as mais diversas bibliotecas e frameworks do mercado. Por exemplo, se quisermos instalar o tds do jQuery, acessamos

<https://www.npmjs.com/package/@types/jquery>

Se quisermos do lodash ou underscore acessamos

https://www.npmjs.com/package/@types/lodash

https://www.npmjs.com/package/@types/underscore

Dessa forma, antes de sair buscando pela internet os arquivos tsd que precisamos, podemos tentar a sorte executando o comando:

npm install @types/nomeDaLibOuFramework --save-dev

Nesse sentido, se quisermos instalar os tds das três bibliotecas que foram citadas, fazemos:

npm install @types/jquery --save-dev

npm install @types/loadash --save-dev

npm install @types/underscore --save-dev

Qualquer tds files que esteja dentro de node\_modules/@types será lidado automaticamente pelo compilador do TypeScript.

É preciso se conformar quando não houver do Typing para sua biblioteca preferia, neste caso, a estratégia do declare var que vimos neste treinamento é uma saída, não muito ideal, mas que permitirá seu código compilar até que você encontre seu tsd.

Não é raro o desenvolvedor querer comentar seu código, seja no estilo livro ou usando uma forma padronizada. No entanto, não faz sentido esses comentários irem para o arquivo de produção, pois além de deixarem evidente para terceiros o que o código que escrevemos faz, ainda acaba impactando no tamanho final do arquivo.

Podemos resolver esse problema facilmente solicitando ao compilador do TypeScript que remova todos os comentários do arquivo JavaScript resultante da compilação do nosso código TypeScript. Para isso, basta adicionar a propriedade "removeComments": true no arquivo tsconfig.json.