02

Guardando negociações

Transcrição

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!