TypeScript – Decorator

As vezes precisamos **lidar com requisitos não funcionais** *(requisitos que não fazem parte da regra de negócio)*, como por exemplo, um teste de performance. Esse tipo de **verificação as vezes é simples de se fazer**, mas **tende a se tornar um método intrusivo** *(vai precisar estar presente nos métodos que são regra de negócio)* e **repetitivo**:

```
public adiciona(): void {

    const t1 = performance.now();

    const negociacao = Negociacao.criaDe(
        this.inputData.value,
        this.inputQuantidade.value,
        this.inputValor.value
    );

    if (!this.ehDiaUtil(negociacao.data)) {
        this.mensagemView
            .update('Apenas negociações em dias úteis s
        return ;
    }

    this.negociacoes.adiciona(negociacao);
    this.limparFormulario();
    this.atualizaView();

    const t2 = performance.now();
    console.log(`Tempo de execução do método adiciona foi de ${(t2 - t1)/1000} segundos`)
}
```

```
public update(model: T): void {
    const t1 = performance.now();

    let template = this.template(model);
    if (this.escapar) {
        template = template
            .replace(/<script>[\s\S]*?<\/script>/, '');
    }
    this.elemento.innerHTML = template;

    const t2 = performance.now();
    console.log(`Tempo de execução do método atualizar foi de ${(t2 - t1)/1000} segundos`)
}
```

A ideia então é usar decorator para centralizar esse código repetitivo e torna-lo disponivel para qualquer lugar da aplicação.

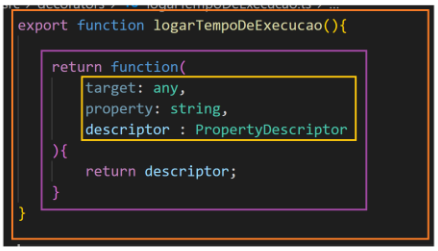
Decorator

Para que os **decorators** funcionem corretamente é necessário **habilitar uma configuração no *"tsconfig.json"*** :

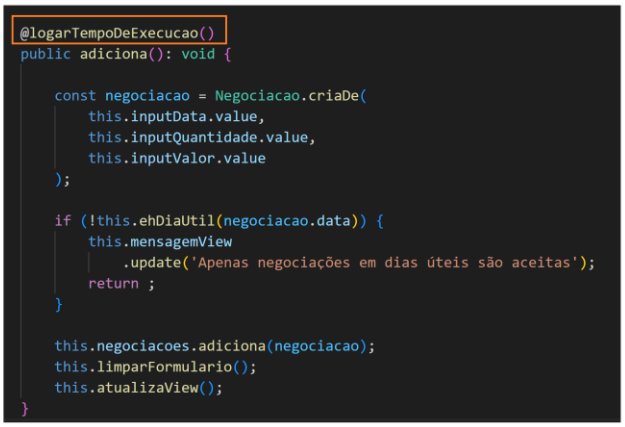
```
tsconfig.json > ...
{
    "compilerOptions": {
        "outDir": "app/dist/js",
        "target": "ES6",
        "noEmitOnError": true,
        "noImplicitAny": true,
        "removeComments": true,
        "strictNullChecks": true,
        "experimentalDecorators": true
    },
    "include": ["app/src/**/*"]
}
```

Um **decorator** nada mais é do que uma **função** que retorna **outra função com parâmetros específicos** e pode ser exportada para ser utilizada em algum lugar. Beleza, mas **o que são os parâmetros específicos?** Eles consistem em 3:

- target: é o nosso alvo, se eu coloco um **decorator** em cima de um método estático, meu **alvo vai ser a função construtora da classe**, se não for estático ele me retorna o **prototype da classe**. Resumindo, ele vai ser **o alvo que contém o método** que eu estou anotando com o decorator**.**
- propertyKey : ele é o nome do método que foi decorado
- descriptor : já o propertyDescriptor vai ter os detalhes do método que eu decorei, ele conhece tudo a respeito do método original.



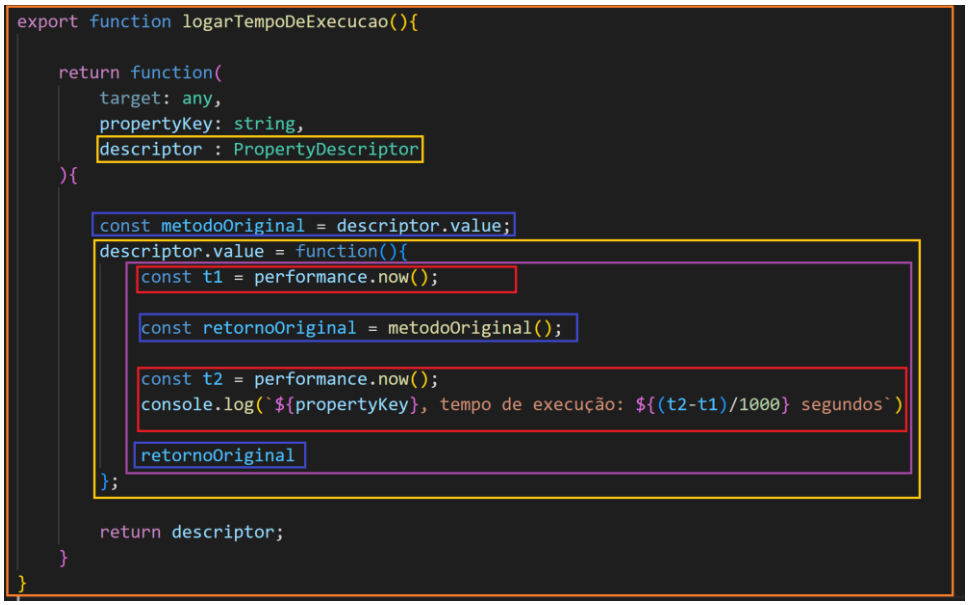
```
export function logarTempoDeExecucao(){

    return function(
        target: any,
        property: string,
        descriptor : PropertyDescriptor
    ){
        return descriptor;
    }
}
```



```
@logarTempoDeExecucao()
public adiciona(): void {

    const negociacao = Negociacao.criaDe(
        this.inputData.value,
        this.inputQuantidade.value,
        this.inputValor.value
    );

    if (!this.ehDiaUtil(negociacao.data)) {
        this.mensagemView
            .update('Apenas negociações em dias úteis são aceitas');
        return ;
    }

    this.negociacoes.adiciona(negociacao);
    this.limparFormulario();
    this.atualizaView();
}
```

Esse é apenas o **esqueleto de um decorator**, pra que ele seja realmente útil precisa de uma implementação. A **implementação** nesse contexto **é saber o tempo de execução** de **determinado método,** esse determinado método é acessível no **decorator** através do **propertyDescriptor** com a propriedade ***"value"***, ou seja, eu posso capturar esse **método original** e sobrescrever seu **comportamento** com a **minha implementação.**

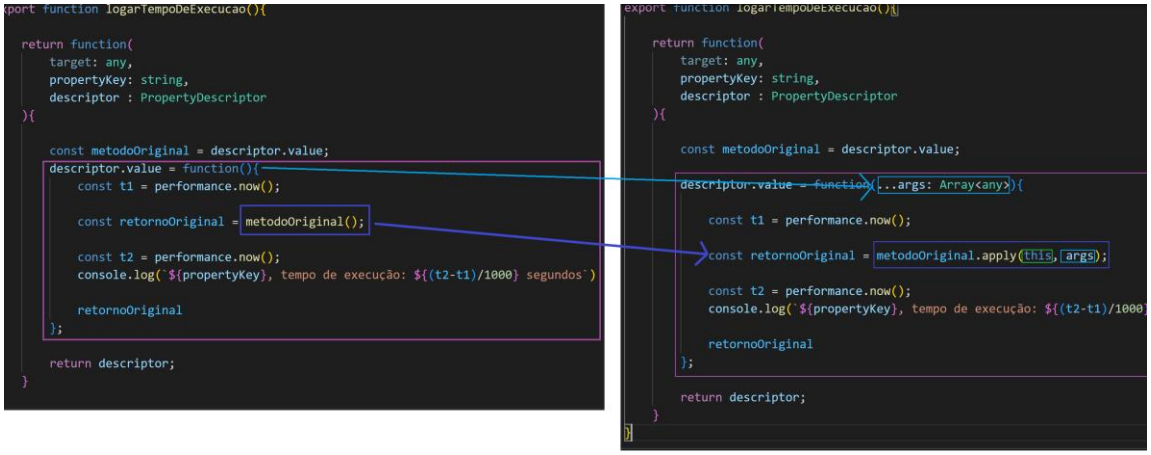
É como se eu colocasse o **meu método original** no **meio da minha implementação de log** e **devolvesse seu valor original de execução**, eu apenas **incrementei o comportamento dele:**



```
export function logarTempoDeExecucao(){

    return function(
        target: any,
        propertyKey: string,
        descriptor : PropertyDescriptor
    ){

        const metodoOriginal = descriptor.value;
        descriptor.value = function(){
            const t1 = performance.now();

            const retornoOriginal = metodoOriginal();

            const t2 = performance.now();
            console.log(`${propertyKey}, tempo de execução: ${(t2-t1)/1000} segundos`)

            retornoOriginal
        };

        return descriptor;
    }
}
```

Só que esse comportamento ainda pode ser melhorado, **minha ideia é medir o tempo de execução de qualquer método** existente na aplicação, dessa forma o **decorator** tem que saber lidar **com métodos que tem N parâmetros ou nenhum,** além disso, o **decorator** precisa saber o **contexto do método original** *(principalmente para referenciar os **"this"** que estão envolvidos).*

Eu posso resolver esse problema dizendo que **meu novo comportamento atribuído** ao valor vai poder **receber um array de argumentos de qualquer tipo**, e meu **método original** será executado com **apply**(que chama a classe de contexto original, assim teremos a referência do ***"this"***)



Com isso eu criei um decorator genérico o bastante para informar a performance de qualquer método.

Decorator – parâmetros default

Decorators também podem receber parâmetros obviamente, se eles forem opcionais podemos usar padrões default, que nada mais **é do que você atribuir ao parâmetro um valor padrão**: