08

Programação defensiva

Transcrição

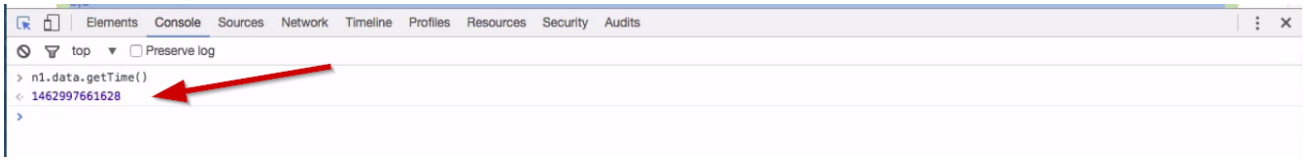
Uma maneira de tornar a instância imutável é quando chamam a propriedade getter `data` e é retornado uma nova instância de `Date` com a mesma data da negociação. Nós devolveremos uma nova referência, um novo objetivo. Atualmente, o `get` do arquivo `Negociacao.js` está assim:

```
get volume() {

    return this._quantidade * this._valor;

}

get data() {
    return this._data;
}

get quantidade() {
    return this._quantidade;
}

get valor() {
    return this._valor;
}
```

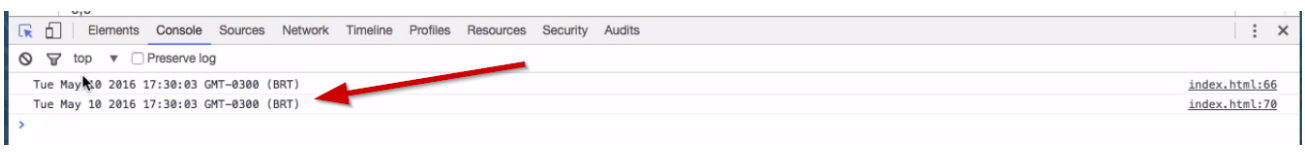
Vamos modificar o retorno do `get data()` :

```
get data() {
    return new Date(this._data.getTime());
}
```

O `getTime` de uma `data` retornará um número *long* com uma representação da data. Se digitarmos no Console `n1.data.getTime()` , ele retornará um número que representará a data:



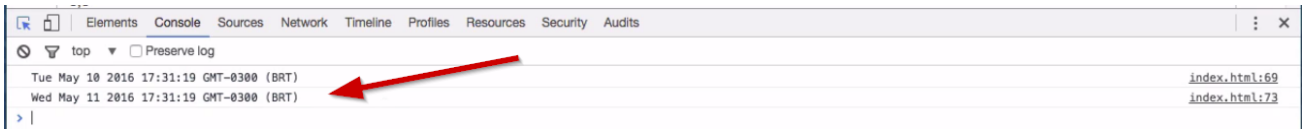
No construtor de `Date()` , este número será aceito para a construção de uma nova data. Então, quando pedimos uma nova `data` , ela será criada baseada na data da negociação. Trata-se de um novo objeto. Se tentarmos alterar no `data` do `index.html` , apenas a cópia será alterada - o novo objeto que retornei `date` , enquanto o interno seguirá inalterado. Isto é o que chamamos de programação defensiva. Vamos testar o nosso código, após as alterações feitas no `get data()` do `Negociacao.js` . Depois de recarregarmos a página no navegador, veremos a seguinte data no Console:



Dessa vez não conseguimos alterar as datas no Console. Isto ocorreu porque apesar de termos usado o `n1.data.setDate(11)`, ele não retornará a data original do objeto que ele já tem. Ele criará um novo objeto, uma nova referência baseada naquela data. Se alteramos o objeto, como fizemos no `setDate()`, não modificaremos a data da negociação. Devemos ter o mesmo cuidado com o construtor. Porque se passamos uma data no construtor do arquivo `index.html`, por exemplo, adicionando uma variável `hoje`.

```
<script>

    var hoje = new Date();

    var n1 = new Negociacao(hoje, 5, 700);

    console.log(n1.data);

    hoje.setDate(11);

    console.log(n1.data);
</script>
```

Observe que em vez de passarmos o `new Date()`, usamos o `hoje`. Quando a `Negociacao` receber o objeto `hoje`, ele guardará uma referência para o mesmo objeto. Isto significa que se alteramos a variável `hoje`, modificaremos a data que está na `Negociacao`. Se executarmos o código, a data será alterada para o dia `11`.



Por isso, usaremos o `getTime()` no `construtor`.

```
class Negociacao {

    construtor(data, quantidade, valor) {

        this._data = new Date(data.getTime());
        this._quantidade = quantidade;
        this._valor = valor;
        Object.freeze(this);

    }
//...
```

Ao fazermos isto, já não conseguiremos alterar a referência no `index.html`, porque o que estamos guardando no `Negociacao` não é mais uma referência para `hoje`, nós usaremos um novo objeto. Então, quando recarregarmos a página no navegador, a data que aparecerá no Console será `10`.

No momento da criação do design das classes, **seja cuidadoso** com a **imutabilidade**.

Agora, você pode estar pensando o seguinte: apesar de `_quantidade` e `_valor` serem números, eles são objetos também. Mas isso não é um problema quando trabalhamos com o `_quantidade`, porque não temos nenhum método que irá alterá-lo. A única forma de fazermos isto é atribuindo um valor. Mas como `_quantidade` é imutável, não conseguiremos realizar esta ação também.

Com isso, finalizamos a blindagem da nossa classe para garantir a sua imutabilidade. Existem outras soluções mais avançadas no JS para tentarmos emular o *privacy* - a privacidade - do seu código. Mas, ao aplicá-las perdemos em legibilidade e performance. Então, a solução utilizada é a mais viável.