

Consolidando seu conhecimento 2 (importante)

Você deve ter estranhado esse exercício vir depois do exercício "Consolidando seu conhecimento 1". Isso não foi por acaso. Apesar de o código apresentado neste capítulo ser totalmente funcional, ele pode falhar em outros cenários que não sejam o dessa aplicação. Queremos um código genérico, não é mesmo? A alteração é ínfima, mas envolve uma revisão do que aprendemos de proxy. Preparado?

Vamos revisitar a classe ProxyFactory :

```
class ProxyFactory {

  static create(objeto, props, acao) {

    return new Proxy(objeto, {

      get(target, prop, receiver) {

        if(props.includes(prop) && ProxyFactory._ehFuncao(target[prop])) {

          return function() {

            console.log(`interceptando ${prop}`);
            Reflect.apply(target[prop], target, arguments);
            acao(target);

          }

        }

        return Reflect.get(target, prop, receiver);
      },

      set(target, prop, value, receiver) {

        if(props.includes(prop)) {
          target[prop] = value;
          acao(target);
        }

        return Reflect.set(target, prop, value, receiver);
      }

    });
  }

  static _ehFuncao(func) {

    return typeof(func) == typeof(Function);

  }
}
```

Com o projeto aberto no Chrome, abra o console e crie um objeto que possui um método que **retorna um valor**. Você pode cortar e colocar o código a seguir:

```
let pessoa = {
  nome: 'Flávio',
  sobrenome: 'Almeida',
  getNomeCompleto() {
    return `${this.nome} ${this.sobrenome}`;
  }
}
```

Se quisermos obter o nome completo fazemos `pessoa.getNomeCompleto()` o que exibirá no console a mensagem "Flávio Almeida". Perfeito.

Agora vamos criar um proxy desse objeto. Ainda com o Chrome aberto, cole o seguinte código no seu terminal:

```
xy = ProxyFactory.create(pessoa, ['getNomeCompleto'], () => console.log('armadilha aqui'));
```

Criamos nosso proxy! Agora vamos chamar o método `pessoaProxy.getNomeCompleto()`. O que deve acontecer? Três saídas devem acontecer. A primeira, da `ProxyFactory` que indica o que esta sendo interceptado, a segunda é `armadilha aqui` que nós definimos. Por fim, o retorno de `getNomeCompleto` que deve ser `Flávio Almeida`. Faça um teste e veja algo curioso:

```
pessoaProxy.getNomeCompleto();
```

No lugar de exibir `Flávio Almeida`, o resultado é `undefined`! Isso acontece, porque em nossa `ProxyFactory`, quando interceptamos um método, não estamos fazendo com que o valor resultante da chamada do método seja retornado! Resumindo: do jeito que está, métodos com retorno de proxies criadas a partir da `ProxyFactory` retornarão sempre `undefined`! E agora?

A correção é simples. Vamos na parte do nosso código que identificamos que a propriedade é uma função. Vou colocar apenas esse trecho de código:

```
// ProxyFactory.js
// código anterior omitido

if(props.includes(prop) && ProxyFactory._ehFuncao(target[prop])) {

  return function() {

    console.log(`interceptando ${prop}`);
    Reflect.apply(target[prop], target, arguments);
    acao(target);
  }
}
// código posterior omitido
```

Veja que em nenhum momento retornamos o resultado de `Reflect.apply(target[prop], target, arguments)`. Não podemos simplesmente colocar um `return` na frente dessa instrução porque assim `acao(target)` nunca será executado. Vamos guardar o retorno em uma variável, chamar `acao(target)` e aí sim retornar o resultado da operação:

```
// ProxyFactory.js
// código anterior omitido
```

```

if(props.includes(prop) && ProxyFactory._ehFuncao(target[prop])) {

    return function() {

        console.log(`interceptando ${prop}`);
        let retorno = Reflect.apply(target[prop], target, arguments);

        acao(target);
        return retorno;
    }
}
// código posterior omitido

```

Se por acaso o método não retornar nada, não faz mal, o retorno será `undefined`, algo totalmente esperado.

Agora, recarregue a página `index.html` mais uma vez cole o código abaixo para realizar um novo teste:

```

let pessoa = {
    nome: 'Flávio',
    sobrenome: 'Almeida',
    getNomeCompleto() {
        return `${this.nome} ${this.sobrenome}`;
    }
}

let pessoaProxy = ProxyFactory.create(pessoa, ['getNomeCompleto'], () => console.log('armadilha'))

pessoaProxy.getNomeCompleto();

```

Agora sim! Interceptamos o método e o seu retorno agora é disponível para quem o chamou. Como nenhum dos métodos interceptados da nossa aplicação retornavam um valor, a ausência dessa mudança não impactava no resultado final. Mas como disse, queremos uma solução genérica que possa ser usada em qualquer situação, inclusive para métodos que retornam valor.

Por fim, há ainda uma pequena alteração que envolve mais performance. Quando interceptamos a escrita em uma propriedade, nosso handler `set` é executado. Vejamos seu código:

```

// ProxyFactory.js
// código anterior omitido
set(target, prop, value, receiver) {

    if(props.includes(prop)) {
        target[prop] = value;
        acao(target);
    }
    return Reflect.set(target, prop, value, receiver);
}
// código posterior omitido

```

O código funciona, mas um olhar atento percebe que se a propriedade é uma que estamos monitorando, aplicamos `target[prop] = value` para aplicar o valor recebido na propriedade. Mas veja que precisamos fazer a mesma coisa se a

propriedade não é monitorada, caso contrário ela nunca receberá seu valor. É por isso que logo em seguida realizamos `return Reflect.set(target, prop, value, receiver)`. Veja que há um `return` porque uma atribuição em uma propriedade `setter` pode retornar um valor, apesar de isso não ser comum. Sendo assim, atualizamos o objeto original encapsulado duas vezes quando ele possui uma propriedade que queremos interceptar e executar uma armadilha.

Otimizando nosso código:

```
// ProxyFactory.js
// código anterior omitido


set(target, prop, value, receiver) {

    let retorno = Reflect.set(target, prop, value, receiver);
    if(props.includes(prop)) {
        acao(target);
    }
    return retorno;
}
// código posterior omitido
```

Se quiser, ainda podemos remover o bloco do `if` :

```
// ProxyFactory.js
// código anterior omitido
set(target, prop, value, receiver) {

    let retorno = Reflect.set(target, prop, value, receiver);
    if(props.includes(prop)) acao(target);    // só executa acao(target) se for uma proprie
    return retorno;
}
// código posterior omitido
```



Agora a `ProxyFactory` está ainda mais redondinha!

Por último implemente a classe `Bind`. Ela receberia três parâmetros apenas: o modelo, as propriedades que desejamos monitorar e a view. Não se esqueça de utilizar os parâmetros `REST`.