JS-12: JavaScript Moderno

Aula 4: OOP, Prototypes e JSON

OOP: Programação Orientada a Objetos com JavaScript

Até o momento, trabalhamos somente com **objetos literais** no JS. Isso significa que quando precisávamos criar um objeto, não era necessário definir seu tipo. Bastava somente criar uma constante (ou uma variável) e atribuir a ela uma sequência de propriedades e métodos que seriam necessários. Veja:

```
const objeto = {
        propriedade1 : "valor da propriedade #1",
        propriedade2 : "valor da propriedade #2",
        metodo1 : function() {
                console.log("Coisa que o método 1 faz…");
        },
        metodo2 : function() {
                console.log("Coisa que o método 2 faz…");
        }
}
```

Por exemplo, vamos imaginar que precisamos guardar as informações de uma **Notícia**. Quais dados precisaríamos? Poderíamos ter: *título da notícia*, *resumo da notícia*, *texto completo da notícia*, *data de publicação* e *autor da notícia*.

Veja um exemplo da estrutura mencionada abaixo:

```
const noticia = {
        titulo : "Ministério da Educação abre inscrições para o ENEM",
        resumo : "Inscrições irão iniciar mais cedo esse ano",
        textoCompleto : "Suspendisse sed est ut risus maximus ultricies non vitae metus.
        Aliquam dignissim odio velit, id dapibus risus ornare non. Cras nec pulvinar leo.",
        dataPublicacao : "15 de Abril de 2019",
        autor : "João Cunha Andrade"
}
```

Pronto, temos a nossa notícia estruturada num objeto com as propriedades que precisamos. Porém, como estamos trabalhando com um objeto literal, ele é livre para conter as

propriedades que quisermos. Infelizmente, não temos aqui como garantir que uma notícia sempre será gerada com exatamente essas propriedades.

É nesse momento que se faz necessário trabalhar com tipos de dados mais especializados. Aqui, para garantirmos (ou ao menos aumentarmos as chances) de uma notícia ter ao menos essas informações de base, o ideal é criarmos um **tipo customizado**, ou seja, um objeto do tipo **Notícia**.

Até a versão ES5 do JavaScript, isso se dava com o uso das chamadas **funções construtoras**. Essas funções guardam dentro de si a estrutura de propriedades que desejamos criar, e ao serem chamadas em conjunto com o operador **new**, criam um objeto com as propriedades e demais características que elas foram programadas para realizar.

Veja:

```
// função construtora do tipo "Noticia"
function Noticia() {
    this.titulo = "";
    this.resumo = "";
    this.texto = "";
    this.dataPublicacao = "";
    this.autor = "";
}

// Pedimos ao JS que crie um novo objeto com todas as características
// da função "Noticia". Ou seja, ele será um objeto do tipo Noticia.
// Assim garantimos que todas as propriedades básicas necessárias sempre estarão
// disponíveis para nós
const noticia1 = new Noticia();
noticia1.titulo = "Minha notícia #1";
noticia2.resumo = "Resumo da notícia #1";

// outra cópia de Noticia()
const noticia2 = new Noticia();
```

O legal é que o objeto gerado, apesar de continuar sendo do tipo "object", agora ele representa uma cópia de "*Noticia"*, isto é, ele é uma **instância** de "*Noticia"* e isso pode ser usado para realizar validações.

Vamos imaginar que um programador da nossa equipe tenha criado um módulo que gerencia notícias no banco de dados. Uma das funções do módulo que ele criou foi a "adicionarNoticia" que recebe um objeto "Noticia" para ser cadastrado no banco de dados. Veja abaixo:

```
export function adicionarNoticia( noticia ) {
    // … lógica para adicionar a notícia no banco ...
}
```

OK!
Mas como ele pode ter certeza que o parâmetro recebeu uma *“Noticia”* e não um outro objeto qualquer? Lembre-se que em JavaScript podemos criar objetos literais, ou seja, objetos de forma livre e isso não impediria que alguém executasse essa função passando um objeto com propriedades diferentes da esperada.

Usar o operador **typeof** para checar o tipo não ajudaria, isso porque tanto uma notícia real quanto um objeto literal são do mesmo tipo “object”.

```
const noticia = new Noticia();
const noticiaFake = { };

console.log(typeof noticia); // “object”
console.log(typeof noticiaFake); // “object”
```

Ao invés de checar o tipo então, teríamos que pedir ao JS que verifique se a notícia recebida via parâmetro na função *“é uma cópia”* de *“Notícia”*, ou seja, se ela é uma instância de *“Notícia”* e para isso usamos o operador **instanceof**, veja:

```
const noticia = new Noticia();
const noticiaFake = { };

console.log(noticia instanceof Noticia); // true
console.log(noticiaFake instanceof Noticia); // false
```

Agora ficou fácil! Se a notícia recebida para ser adicionada na base de dados NÃO FOR uma instância de “Noticia”, basta lançarmos um erro para alertar o programa que a notícia passada é inválida. Assim nosso código fica mais protegido contra potenciais erros de envio de informações.

Vejamos como ficaria a função do módulo:

```
export function adicionarNoticia( noticia ) {
    if (noticia instanceof Noticia === false) {
        // programa lança um erro e para a execução da função
        throw new Error(“Notícia informada é inválida!”);
    }
    // … lógica para adicionar a notícia no banco ...
}
```

ES6 Classes vs Funções Construtoras
Escrever uma função para criar tipos distintos de objetos não é muito intuitivo. Principalmente para programadores que já tiveram contato com linguagens orientadas a objetos como o Java e o C#.

Nessas linguagens, quando queremos construir tipos de objetos customizados não utilizamos funções para definir suas propriedades e métodos, mas sim **CLASSES** e é isso agora que o ES6 trouxe de novidade: podemos criar objetos por meio da definição de classes.

Com a nova sintaxe do ES6 podemos definir o tipo notícia dessa forma agora:

```
class Noticia {
    constructor() {
        this.titulo = "";
        this.resumo = "";
        this.texto = "";
        this.dataPublicacao = "";
        this.autor = "";
    }
}
```

Com essa nova sintaxe, programadores que trabalharam (ou ainda trabalham) com outras linguagens orientadas a objetos podem se sentir mais familiarizados com o código.

Por debaixo dos panos, o JS continua criando funções construtoras e tals, mas para os programadores que vêm de outras linguagens acabou facilitando mais o entendimento de como objetos são criados no JS já que agora o ES6 faz com que sua sintaxe fique muito próxima a de outras linguagens do mercado tornando menos incômoda a transição entre uma linguagem e outra (muito comum hoje em dia).

Prototypes: entendendo melhor como o JavaScript cria e mantém Objetos.

O JavaScript, ao definir um tipo de objeto, atribui a ele uma propriedade chamada *"prototype"*. Essa propriedade será utilizada pelo JavaScript para compartilhar recursos que deverão estar presentes de forma comum em todas as cópias (instâncias) do tipo criado.

Meio complicado, né? Calma lá! Vai ficar mais claro.

Vamos imaginar a seguinte função construtora:

```
function Pessoa() {
        this.nome = “”;
        this.sobrenome = “”;
        this.mostrarNomeCompleto = function() {
                console.log(this.nome + “ “ + this.sobrenome);
        }
}
```

Ao criarmos objetos a partir dela, teremos cópias exatas da estrutura de propriedades e métodos dessa função nos objetos resultantes. Ou seja, cada objeto criado a partir dessa função terá uma cópia própria das propriedades e métodos definidos nela.

```
// contém: nome, sobrenome e método mostrarNomeCompleto
const pessoa1 = new Pessoa();

// também contém uma cópia zerada das mesmas propriedades
const pessoa2 = new Pessoa();
```

Contudo, lembre-se que algumas funcionalidades estarão presentes em todos os objetos criados a partir da função construtora e que seu comportamento não varia de objeto para objeto. Por exemplo, a função *mostraNomeCompleto()* sempre acessa as propriedades nome e sobrenome do objeto que a executou. Esse sempre será seu comportamento! Sendo assim, não faz sentido termos uma cópia dessa função para cada objeto criado. As propriedades nome e sobrenome sim devem ser copiadas e existir para cada objeto criado já que uma pessoa pode ter um nome e um sobrenome diferente de outras. Mas a função *mostraNomeCompleto()*, sempre fará a mesma coisa: pegar os valores das propriedades do objeto do contexto atual e exibir no console.

Logo, essa função pode ser uma função compartilhada entre todos os objetos pessoa, isto é, não é necessário ter diversas cópias dela para cada objeto criado. Basta que exista somente uma e que cada objeto “Pessoa” a use quando precisar. Um único recurso compartilhado por todos os objetos daquele “tipo”. É aí que entra o *prototype*.

Se ao invés de colocarmos a função mostrarNomeCompleto() dentro da função ou classe construtora, a colocássemos como parte do *prototype* do tipo “Pessoa” isso indica para o JS que a função mostrarNomeCompleto() passa a ser um recurso compartilhado entre todos os objetos do tipo “Pessoa”. Como esse é um recurso compartilhado isso indica que a função só existe em um único lugar (no *prototype* de “Pessoa”), e que todos os objetos da aplicação apontam para essa função (eles mantêm uma referência interna para ela) através da propriedade *__proto__*. Isso é um recurso super interessante que além de economizar memória faz com que algo associado ao *prototype* de um objeto esteja disponível automaticamente para todas as instâncias daquele tipo.

Veja:

```
Pessoa.prototype.mostrarNomeCompleto = function() {
        console.log(this.nome + " " + this.sobrenome);
}
```

Agora, todos os objetos instâncias de "Pessoa" que já foram criados ou que serão criados, terão a função *mostrarNomeCompleto()* já disponível para ser utilizada não precisando de um novo instanciamento para isso.

Podemos estender outras funcionalidades de objetos do próprio JS através do *prototype*. Por exemplo:

```
Array.prototype.removerItem = function(indice) {
        this.splice(indice, 1);
}

const lista = ['Arroz', 'Mamão', 'Manga', 'Feijão'];
lista.removerItem(2); // qualquer array no JS passa a ter a função 'removerItem()'
```

Obs.: Embora possamos fazer o que foi mostrado acima, não é recomendada a alteração do prototype de tipos nativos no JS. Podemos fazer, mas apenas em casos muito, muito especiais.

JSON.parse() e JSON.stringify()

Compartilhar, guardar ou transferir dados pela internet entre aplicações é uma necessidade constante no dia a dia de desenvolvimento de sistemas. Há diversos formatos para a realização dessas transferências ou armazenamentos, porém um dos mais comuns hoje em dia é o formato JSON (JavaScript Object Notation).

Quando consultamos uma API é muito comum que ela retorne os dados necessários nesse formato que nada mais é que uma representação de um objeto JavaScript em formato de *string*.

Veja:

```
const dadosJson = '{"nome" : "Paula", "sobrenome" : "Oliveira"}';
```

Repare que temos um objeto literal (**{ }**) dentro de uma string (**' '**). Isso é necessário pois para transferirmos dados pela internet eles precisam estar em formato texto. Todavia, ao

recebermos esses dados em nossa aplicação JavaScript como texto ele acaba sendo inútil pois não temos como acessar as informações de forma separada (por exemplo, exibir somente a propriedade **nome** do json retornado).

Para que esses dados se tornem mais úteis, temos que convertê-los em um objeto JS real. É aí que as funções do objeto JSON do JavaScript entram em cena.

- **JSON.parse(***jsonEmString***):** converte um json ainda em formato string (texto) em um objeto ou tipo literal do JavaScript.

```
const dadosJson = '{"nome" : "Paula", "sobrenome" : "Oliveira"}';
alert(dadosJson.nome); // undefined: os dados ainda estão em formato string

const dadosJsonObjeto = JSON.parse(dadosJson);
alert(dadosJsonObjeto.nome); // "Paula"
```

- **JSON.stringify(***jsonEmObjeto***):** converte um objeto ou tipo literal do JavaScript em string (texto json) para ser armazenado ou transferido pela internet.

```
const objetoJS = { nome : "Paula", sobrenome : "Oliveira" };
const objetoStringJson = JSON.stringify(objetoJS);
alert(objetoStringJson); // ' { "nome" : "Paula", "sobrenome" : "Oliveira" } '
```

Assim, quando precisarmos transformar uma STRING JSON em um OBJETO ou TIPO LITERAL do JS, usamos *JSON.parse()*.

Quando precisarmos fazer o oposto, ou seja, transformar um OBJETO ou TIPO LITERAL do JS em uma STRING JSON, usamos *JSON.stringify()*.