

OBJETOS

Constructor Functions

Objetos

Criar um objeto é simples, basta definirmos uma variável e iniciar a definição do seu valor com chaves `{}`. Mas e se precisarmos criar um novo objeto, com as mesmas características do anterior? É possível com o `Object.create`, mas veremos ele mais tarde.

```
const carro = {  
    marca: 'Marca',  
    preco: 0,  
}  
  
const honda = carro;  
honda.marca = 'Honda';  
honda.preco = 4000;  
  
const fiat = carro;  
fiat.marca = 'Fiat';  
fiat.preco = 3000;
```

*carro, fiat e honda apontam para
o mesmo objeto.*

Constructor Functions

Para isso existem as Constructor Functions, ou seja, Funções Construtoras que são responsáveis por construir novos objetos sempre que chamamos a mesma.

```
function Carro() {  
    this.marca = 'Marca';  
    this.preco = 0;  
}  
  
const honda = new Carro();  
honda.marca = 'Honda';  
honda.preco = 4000;  
const fiat = new Carro();  
fiat.marca = 'Fiat';  
fiat.preco = 3000;
```

*Usar Pascal Case, ou seja,
começar com letra maiúscula.*

new Keyword

A palavra chave `new` é responsável por criar um novo objeto baseado na função que passarmos a frente dela.

```
const honda = new Carro();

// 1 Cria um novo objeto
honda = {};

// 2 Define o protótipo
honda.prototype = Carro.prototype;

// 3 Aponta a variável this para o objeto
this = honda;

// 4 Executa a função, substituindo this pelo objeto
honda.marca = 'Marca';
honda.preco = 0;

// 5 Retorna o novo objeto
return honda = {
  marca: 'Marca'.
```



Parâmetros e Argumentos

Podemos passar argumentos que serão utilizados no momento da criação do objeto.

```
function Carro(marca, preco) {  
    this.marca = marca;  
    this.preco = preco;  
}  
  
const honda = new Carro('Honda', 4000);  
const fiat = new Carro('Fiat', 3000);
```

this Keyword

O **this** faz referência ao próprio objeto construído com a Constructor Function.

```
function Carro(marca, precoInicial) {  
    const taxa = 1.2;  
    const precoFinal = precoInicial * taxa;  
    this.marca = marca;  
    this.preco = precoFinal;  
    console.log(this);  
}  
  
const honda = new Carro('Honda', 2000);
```

*Variáveis dentro da Constructor
estão "protetidas".*

Exemplo Real

Quando mudamos a propriedade seletor, o objeto Dom irá passar a selecionar o novo seletor em seus métodos.

```
const Dom = {
  seletor: 'li',
  element() {
    return document.querySelector(this.seletor);
  },
  ativo() {
    this.element().classList.add('ativo');
  },
}

Dom.ativo(); // adiciona ativo ao li
Dom.seletor = 'ul';
Dom.ativo(); // adiciona ativo ao ul
```

Constructor Function Real

Um objeto criado com uma Constructor, não irá influenciar em outro objeto criado com a mesma Constructor.

```
function Dom() {  
  this.seletor = 'li';  
  const element = document.querySelector(this.seletor);  
  this.ativo = function() {  
    element.classList.add('ativo');  
  };  
}  
  
const lista = new Dom();  
lista.seletor = 'ul';  
lista.ativo();  
  
const lastLi = new Dom();  
lastLi.seletor = 'li:last-child';  
lastLi.ativo();
```

Lembre-se de usar parâmetros

```
function Dom(seletor) {  
    const element = document.querySelector(seletor);  
    this.ativo = function(classe) {  
        element.classList.add(classe);  
    };  
}  
  
const lista = new Dom('ul');  
lista.ativo('ativo');  
  
const lastLi = new Dom('li:last-child');  
lastLi.ativo('ativo');
```

Exercícios

// Transforme o objeto abaixo em uma Constructor Function
const pessoa = {

 nome: 'Nome pessoa',

 idade: 0,

 andar() {

 console.log(this.nome + ' andou');

 }

}

// Crie 3 pessoas, João - 20 anos,

// Maria - 25 anos, Bruno - 15 anos

// Crie uma Constructor Function (Dom) para manipulação

// de listas de elementos do dom. Deve conter as seguintes

// propriedades e métodos:

//

// elements, retorna NodeList com os elementos selecionados

// addClass(classe), adiciona a classe a todos os elementos

// removeClass(classe), remove a classe a todos os elementos

OBJETOS

Prototype

Prototype

A propriedade prototype é um objeto adicionado a uma função quando a mesma é criada.

```
function Pessoa(nome, idade) {  
    this.nome = nome;  
    this.idade = idade;  
}  
  
const andre = new Pessoa('André', 28);  
  
console.log(Pessoa.prototype); // retorna o objeto  
console.log(andre.prototype); // undefined
```

funcao.prototype

É possível adicionar novas propriedades e métodos ao objeto prototype.

```
Pessoa.prototype.andar = function() {  
    return this.nome + ' andou';  
}  
  
Pessoa.prototype.nadar = function() {  
    return this.nome + ' nadou';  
}  
  
console.log(Pessoa.prototype); // retorna o objeto
```

Acesso ao Protótipo

O objeto criado utilizando o construtor, possui acesso aos métodos e propriedades do protótipo deste construtor. Lembrando, `prototype` é uma propriedade de funções apenas.

```
const andre = new Pessoa('André', 28);

andre.nome;
andre.idade;
andre.andar();
andre.nadar();
```

proto

O novo objeto acessa os métodos e propriedades do protótipo através da propriedade `__proto__`. É papel da engine fazer essa busca, não devemos falar com `__proto__` diretamente.

```
// Acessam o mesmo método
// mas __proto__ não terá
// acesso ao this.nome
andre.andar();
andre.__proto__.andar();
```

Herança de Protótipo

O objeto possui acesso aos métodos e propriedades do protótipo do construtor responsável por criar este objeto. O objeto abaixo possui acesso a métodos que nunca definimos, mas são herdados do protótipo de Object.

```
Object.prototype;  
andre.toString();  
andre.isPrototypeOf();  
andre.valueOf();
```

Construtores Nativos

Objetos, Funções, Números, Strings e outros tipos de dados são criados utilizando construtores. Esses construtores possuem um protótipo com propriedades e métodos, que poderão ser acessadas pelo tipo de dado.

```
const pais = 'Brasil';
const cidade = new String('Rio');

pais.charAt(0); // B
cidade.charAt(0); // R

String.prototype;
```

É possível acessar a função do protótipo

É comum, principalmente em códigos mais antigos, o uso direto de funções do protótipo do construtor Array.

```
const lista = document.querySelectorAll('li');

// Transforma em uma array
const listaArray = Array.prototype.slice.call(lista);
```

| Existe o método `Array.from()`

Método do Objeto vs Protótipo

Nos objetos nativos existem métodos linkados diretamente ao Objeto e outros linkados ao protótipo.

```
Array.prototype.slice.call(lista);
Array.from(lista);

// Retorna uma lista com os métodos / propriedades
Object.getOwnPropertyNames(Array);
Object.getOwnPropertyNames(Array.prototype);
```

`dado.constructor.name`, retorna
o nome do construtor;

Apenas os Métodos do Protótipo são Herdados

```
[1,2,3].slice(); // existe  
[1,2,3].from(); // não existe
```

Entenda o Que está Sendo Retornado

Os métodos e propriedades acessado com o `.` são referentes ao tipo de dados que é retornado antes desse `.`

```
const Carro = {  
    marca: 'Ford',  
    preco: 2000,  
    acelerar() {  
        return true;  
    }  
}  
  
Carro // Object  
Carro.marca // String  
Carro.preco // Number  
Carro.acelerar // Function  
Carro.acelerar() // Boolean  
Carro.marca.charAt // Function  
Carro.marca.charAt(0) // String
```

Verifique o nome do construtor

ORIGAMID

Exercícios

```
// Crie uma função construtora de Pessoas  
// Deve conter nome, sobrenome e idade  
// Crie um método no protótipo que retorne  
// o nome completo da pessoa  
  
// Liste os métodos acessados por  
// dados criados com NodeList,  
// HTMLCollection, Document  
  
// Liste os construtores dos dados abaixo  
const li = document.querySelector('li');  
  
li;  
li.click;  
li.innerText;  
li.value;  
li.hidden;  
li.offsetLeft;
```

```
// Qual o construtor do dado abaixo:  
li.hidden.constructor.name;
```

OBJETOS

Native, Host e User

Native

Objetos nativos são aqueles definidos na especificação da linguagem e são implementados independente do host.

```
// Construtores de objetos nativos
Object
String
Array
Function
```

Host

Objetos do host são aqueles implementados pelo próprio ambiente. Por exemplo no browser possuímos objetos do DOM, como DomList, HTMLCollection e outros. Em Node.js os objetos do Host são diferentes, já que não estamos em um ambiente do browser.

```
// Objetos do browser
NodeList
HTMLCollection
Element
```

User

Objetos do user, são os objetos definidos pelo seu aplicativo. Ou seja, qualquer objeto que você criar ou que importar de alguma biblioteca externa.

```
const Pessoa = {  
    nome: 'André';  
}
```

Diferentes Versões

- **Browsers diferentes**

Apesar de tentarem ao máximo manter um padrão, browsers diferentes possuem objetos com propriedades e métodos diferentes.

- **Versões de browsers**

Sempre que o browser é atualizado, novos objetos, métodos e propriedades podem ser implementados.

- **Host e Native Objects**

Por exemplo, browsers que não implementaram o ECMAScript 2015 (ES6), não possuem o método `find` de `Array`.

Versões de JavaScript

- **ECMA**

Organização responsável por definir padrões para tecnologias. ECMAScript é o padrão de JavaScript.

- **ECMAScript 2015 ou ES6**

ES é uma abreviação de ECMAScript, ES6 é a sexta versão do ECMAScript, que foi lançada em 2015. Por isso ECMAScript 2015 é igual a ES6. A partir da ES6, existe uma tendência anual de atualizações. ECMAScript 2015, 2016, 2017, 2018 e Next.

- **Engine**

Existem diversas engines que implementam o ECMAScript como V8, SpiderMonkey, Chakra, JavaScriptCore e mais.

Bibliotecas

Bibliotecas como jQuery foram criadas para resolver o problema de inconsistências entre browsers e adicionar funcionalidades que não existiam nativamente. A padronização dos browsers e a implementação de soluções nativas, torna as mesmas obsoletas.

```
$('a').addClass('ativo');  
$('a').hide();  
$('a').show();
```

Verificar se Existe

O `typeof` retorna o tipo de dado. Caso esse dado não exista ou não tenha sido definido, ele irá retornar `undefined`. Ou seja, quando não for `undefined` quer dizer que existe.

```
if (typeof Array.from !== "undefined")
if (typeof NodeList !== "undefined");
```

API

Application Programming Interface, é uma interface de software criada para a interação com outros softwares.

Ou seja, toda interação que fazemos com o browser utilizando Objetos, Métodos e Propriedades, estamos na verdade interagindo com a API do browser.

Exercícios

// Liste 5 objetos nativos

// Liste 5 objetos do browser

// Liste 2 Métodos, Propriedades ou Objetos

// presentes no Chrome que não existem no Firefox

OBJETOS

String

String

É a construtora de strings, toda string possui as propriedades e métodos do prototype de String.

```
const comida = 'Pizza';
const liquido = new String('Água');
const ano = new String(2018);
```

str.length

Propriedade com o total de caracteres da string.

```
const comida = 'Pizza';
const frase = 'A melhor comida';

comida.length; // 5
frase.length; // 15

comida[0] // P
frase[0] // A
frase[frase.length - 1] // a
```

str.charAt(n)

Retorna o caracter de acordo com o index de (n).

```
const linguagem = 'JavaScript';

linguagem.charAt(0); // J
linguagem.charAt(2); // v
linguagem.charAt(linguagem.length - 1); // t
```

str.concat(str2, str3, ...)

Concatena as strings e retorna o resultado.

```
const frase = 'A melhor linguagem é ';
const linguagem = 'JavaScript';

const fraseCompleta = frase.concat(linguagem, ' !!!');
```

str.includes(search, position)

Procura pela string exata dentro de outra string. A procura é case sensitive.

```
const fruta = 'Banana';
const listaFrutas = 'Melancia, Banana, Laranja';

listaFrutas.includes(fruta); // true
fruta.includes(listaFrutas); // false
```

str.endsWith(search) e str.startsWith(search)

Procura pela string exata dentro de outra string. A procura é case sensitive.

```
const fruta = 'Banana';

fruta.endsWith('nana'); // true
fruta.startsWith('Ba'); // true
fruta.startsWith('na'); // false
```

str.slice(start, end)

Corta a string de acordo com os valores de start e end.

```
const transacao1 = 'Depósito de cliente';
const transacao2 = 'Depósito de fornecedor';
const transacao3 = 'Taxa de camisas';

transacao1.slice(0, 3); // Dep
transacao2.slice(0, 3); // Dep
transacao3.slice(0, 3); // Tax

transacao1.slice(12); // cliente
transacao1.slice(-4); // ente
transacao1.slice(3, 6); // ósi
```

str.substring(start, end)

Corta a string de acordo com os valores de start e end. Não funcionar com valores negativos como o slice.

```
const linguagem = 'JavaScript';
linguagem.substring(3,5); // aS
linguagem.substring(0,4); // Java
linguagem.substring(4); // Script
linguagem.substring(-3); // JavaScript
```

str.indexOf(search) e str.lastIndexOf(search)

Retorna o index da string, assim que achar o primeiro resultado ele já retorna. No caso do lastIndexOf ele retorna o último resultado.

```
const instrumento = 'Guitarra';

instrumento.indexOf('r'); // 5
instrumento.lastIndexOf('r'); // 6
instrumento.indexOf('ta'); // 3
```

str.padStart(n, str) e str.padEnd(n, str)

Aumenta o tamanho da string para o valor de n. Ou seja, uma string com 8 caracteres, se passarmos n = 10, ela passará a ter 10 caracteres. O preenchimento é feito com espaços, caso não seja declarado o segundo argumento.

```
const listaPrecos = ['R$ 99', 'R$ 199', 'R$ 12000'];

listaPrecos.forEach((preco) => {
  console.log(preco.padStart(10, '.'));
})

listaPrecos[0].padStart(10, '.'); // .....R$ 99
listaPrecos[0].padEnd(10, '.'); // R$ 99.....
```

str.repeat(n)

Repete a string (n) vezes.

```
const frase = 'Ta';

frase.repeat(5); // TaTaTaTaTa
```

str.replace(regexp|substr, newstr|function)

Troca parte da string por outra. Podemos utilizar uma regular expression ou um valor direto. Se usarmos um valor direto ele irá trocar apenas o primeiro valor que encontrar.

```
let listaItens = 'Camisas Bonés Calças Bermudas Vestidos  
Saias';  
listaItens = listaItens.replace(/\s+/g, ', ');  
  
let preco = 'R$ 1200,43';  
preco = preco.replace(',', '.'); // 'R$ 1200.43'
```

Veremos mais sobre Regular Expression

str.split(padrao)

Divide a string de acordo com o padrão passado e retorna uma array.

```
const listaItens = 'Camisas Bonés Calças Bermudas Vestidos  
Saias';  
const arrayItens = listaItens.split(' ');  
  
const htmlText = '<div>O melhor item</div><div>A melhor  
lista</div>';  
const htmlArray = htmlText.split('div');  
const htmlNovo = htmlArray.join('section');
```

| *join é um método de Array*

str.toLowerCase() e str.toUpperCase()

Retorna a string em letras maiúsculas ou minúsculas. Bom para verificarmos input de usuários.

```
const sexo1 = 'Feminino';
const sexo2 = 'feminino';
const sexo3 = 'FEMININO';

(sexo1.toLowerCase() === 'feminino'); // true
(sexo2.toLowerCase() === 'feminino'); // true
(sexo3.toLowerCase() === 'feminino'); // true
```

str.trim(), str.trimStart(), str.trimEnd()

Remove espaço em branco do início ou final de uma string.

```
const valor = '  R$ 23.00  '
valor.trim(); // 'R$ 23.00'
valor.trimStart(); // 'R$ 23.00  '
valor.trimEnd(); // '  R$ 23.00'
```

Exercícios

```
// Utilizando o foreach na array abaixo,  
// some os valores de Taxa e os valores de Recebimento  
  
const transacoes = [  
  {  
    descricao: 'Taxa do Pão',  
    valor: 'R$ 39',  
  },  
  {  
    descricao: 'Taxa do Mercado',  
    valor: 'R$ 129',  
  },  
  {  
    descricao: 'Recebimento de Cliente',  
    valor: 'R$ 99',  
  },  
  {  
    descricao: 'Taxa do Banco',  
    valor: 'R$ 129',  
  },  
];
```

```
        valor: 'R$ 49',  
    },  
];  
  
// Retorne uma array com a lista abaixo  
const transportes = 'Carro;Avião;Trem;Ônibus;Bicicleta';  
  
// Substitua todos os span's por a's  
const html = `<ul>  
    <li><span>Sobre</span></li>  
    <li><span>Produtos</span></li>  
    <li><span>Contato</span></li>  
</ul>`;  
  
// Retorne o último caracter da frase  
const frase = 'Melhor do ano!';  
  
// Retorne o total de taxas  
const transacoes = ['Taxa do Banco', '    TAXA DO PÃO', '    taxa  
do mercado', 'depósito Bancário', 'TARIFA especial'];
```

OBJETOS

Number e Math

Number

É a construtora de números, todo número possui as propriedades e métodos do prototype de Number. Number também possui alguns métodos.

```
const ano = 2018;  
const preco = new Number(99);
```

Number.isNaN() e Number.isInteger():

isNaN() é um método de Number, ou seja, não faz parte do protótipo. isInteger() verifica se é uma integral.

```
Number.isNaN(NaN); // true  
Number.isNaN(4 + 5); // false
```

```
Number.isInteger(20); // true  
Number.isInteger(23.6); // false
```

Number.parseFloat() e Number.parseInt()

parseFloat() serve para retornarmos um número a partir de uma string. A String deve começar com um número. parseInt recebe também um segundo parâmetro que é o Radix, 10 é para decimal.

```
parseFloat('99.50'); // Mesma função sem o Number  
Number.parseFloat('99.50'); // 99.5  
Number.parseFloat('100 Reais'); // 100  
Number.parseFloat('R$ 100'); // NaN  
  
parseInt('99.50', 10); // 99  
parseInt(5.43434355555, 10); // 5  
Number.parseInt('100 Reais', 10); // 100
```

Float possui decimal, Integer não.

n.toFixed(decimais)

Arredonda o número com base no total de casas decimais do argumento.

```
const preco = 2.99;  
preco.toFixed(); // 3
```

```
const carro = 1000.455;  
carro.toFixed(2) // 1000.46
```

```
const preco2 = 1499.49;  
preco2.toFixed() // 1499
```

n.toString(radix)

Transforma o número em uma string com base no Radix. Use o 10 para o sistema decimal.

```
const preco = 2.99;  
preco.toString(10); // '2.99'
```

n.toLocaleString(lang, options);

Formata o número de acordo com a língua e opções passadas.

```
const preco = 59.49;
preco.toLocaleString('en-US', {style: 'currency', currency:
'USD'}); // $59.49
preco.toLocaleString('pt-BR', {style: 'currency', currency:
'BRL'}); // R$ 59,49
```

Math

É um Objeto nativo que possui propriedades e métodos de expressões matemáticas comuns.

```
Math.PI // 3.14159  
Math.E // 2.718  
Math.LN10 // 2.303
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Math.abs(), Math.ceil(), Math.floor() e Math.round()

abs() retorna o valor absoluto, ou seja, transforma negativo em positivo. ceil() arredonda para cima, retornando sempre uma integral e floor para baixo. round() arredonda para o número integral mais próximo.

```
Math.abs(-5.5); // 5.5
Math.ceil(4.8334); // 5
Math.ceil(4.3); // 5
Math.floor(4.8334); // 4
Math.floor(4.3); // 4
Math.round(4.8334); // 5
Math.round(4.3); // 4
```

Math.max(), Math.min() e Math.random():

max() retorna o maior número de uma lista de argumentos, min() o menor número e random() um número aleatório entre 0 e 1.

```
Math.max(5,3,10,42,2); // 42
Math.min(5,3,10,42,2); // 2

Math.random(); // 0.XXX
Math.floor(Math.random() * 100); // entre 0 e 100
Math.floor(Math.random() * 500); // entre 0 e 500

// Número random entre 72 e 32
Math.floor(Math.random() * (72 - 32 + 1)) + 32;
Math.floor(Math.random() * (max - min + 1)) + min;
```

Exercícios

```
// Retorne um número aleatório  
// entre 1050 e 2000  
  
// Retorne o maior número da lista abaixo  
const numeros = '4, 5, 20, 8, 9';  
  
// Crie uma função para limpar os preços  
// e retornar os números com centavos arredondados  
// depois retorne a soma total  
const listaPrecos = ['R$ 59,99', ' R$ 100,222',  
                    'R$ 230  ', 'r$  200'];
```

OBJETOS

Array

Arrays

Arrays armazenam uma coleção de elementos. Estes podem ser strings, arrays, boolean, number, functions, objects e mais.

```
const instrumentos = ['Guitarra', 'Baixo', 'Violão'];
const precos = [49, 99, 69, 89];

const dados = [new String('Tipo 1'), ['Carro', 'Portas', {cor: 'Azul', preco: 2000}], function andar(nome) { console.log(nome) }];
dados[2]('Ford');
dados[1][2].cor; // azul
```

Construção de Arrays

Toda array herda os métodos e propriedades do protótipo do construtor Array.

```
const instrumentos = ['Guitarra', 'Baixo', 'Violão'];
const carros = new Array('Corola', 'Mustang', 'Honda');

carros[1] // Mustang
carros[2] = 'Ferrari';
carros[10] = 'Parati';
carros.length; // 11
```

Os valores das array's não são estáticos

Array.from()

Array.from() é um método utilizado para transformar array-like objects, em uma array.

```
let li = document.querySelectorAll('li'); // NodeList
li = Array.from(li); // Array

const carros = {
  0: 'Fiat',
  1: 'Honda',
  2: 'Ford',
  length: 4,
}

const carrosArray = Array.from(carros);
```

Array.isArray()

Verifica se o valor passado é uma array e retorna um valor booleano.

```
let li = document.querySelectorAll('li'); // NodeList
Array.isArray(li); // false

li = Array.from(li); // Array
Array.isArray(li); // true
```

Array.of(), Array() e new Array()

Verifica se o valor passado é uma array e retorna um valor booleano. A palavra chave new não é necessária para utilizar o construtor Array.

```
Array.of(10); // [10]
Array.of(1,2,3,4); // [1,2,3,4]
new Array(5); // [,,,,]
Array(5); // [,,,,]
Array(1,2,3,4); // [1,2,3,4]
```

Propriedades e Métodos do Prototype

[].length retorna o tamanho da array.

```
const frutas = ['Banana', 'Pêra', ['Uva Roxa', 'Uva Verde']];
frutas.length; // 3

frutas[0].length; // 6
frutas[1].length; // 5
frutas[2].length; // 2
```

Métodos Modificadores [].sort()

Os próximos métodos que vamos falar sobre, são métodos modificadores (mutator methods). Além de retornarem um valor, eles modificam a array original. `[].sort()` organiza a pelo unicode.

```
const instrumentos = ['Guitarra', 'Baixo', 'Violão'];
instrumentos.sort();
instrumentos; // ['Baixo', 'Guitarra', Violão]
```

```
const idades = [32,21,33,43,1,12,8];
idades.sort();
idades; // [1, 12, 21, 32, 33, 43, 8]
```

[] .unshift() e [] .push()

`[] .unshift()` adiciona elementos ao início da array e retorna o `length` da mesma. `[] .push()` adiciona elementos ao final da array e retorna o `length` da mesma.

```
const carros = ['Ford', 'Fiat', 'VW'];
carros.unshift('Honda', 'Kia'); // 5
carros; // ['Honda', 'Kia', 'Ford', 'Fiat', 'VW'];

carros.push('Ferrari'); // 6
carros; // ['Honda', 'Kia', 'Ford', 'Fiat', 'VW', 'Ferrari'];
```

[] .shift() e [] .pop()

`[] .shift()` remove o primeiro elemento da array e retorna o mesmo. `[] .pop()` remove o último elemento da array e retorna o mesmo.

```
const carros = ['Ford', 'Fiat', 'VW', 'Honda'];
const primeiroCarro = carros.shift(); // 'Ford'
carros; // ['Fiat', 'VW', 'Honda'];

const ultimoCarro = carros.pop(); // 'Honda'
carros; // ['Fiat', 'VW'];
```

[].reverse()

[].reverse() inverte os itens da array e retorna a nova array.

```
const carros = ['Ford', 'Fiat', 'VW', 'Honda'];
carros.reverse(); // ['Honda', 'VW', 'Fiat', 'Ford'];
```

[]splice()

`[].splice(index, remover, item1, item2, ...)` adiciona valores na array a partir do index. Remove a quantidade de itens que for passada no segundo parâmetro (retorna esses itens).

```
const carros = ['Ford', 'Fiat', 'VW', 'Honda'];
carros.splice(1, 0, 'Kia', 'Mustang'); // []
carros; // ['Ford', 'Kia', 'Mustang', 'Fiat', 'VW', 'Honda']

carros.splice(3, 2, 'Ferrari'); // ['Fiat', 'VW']
carros; // ['Ford', 'Kia', 'Mustang', 'Ferrari', 'Honda']
```

[].copyWithin()

`[].copyWithin(alvo, inicio, final)` a partir do alvo, ele irá copiar a array começando do inicio até o final e vai preencher a mesma com essa cópia. Caso omita os valores de início e final, ele irá utilizar como inicio o 0 e final o valor total da array.

```
'Item1', 'Item2', 'Item3', 'Item4'].copyWithin(2, 0, 3);  
// ['Item1', 'Item2', 'Item1', 'Item2']
```

```
'Item1', 'Item2', 'Item3', 'Item4'].copyWithin(-1);  
// ['Item1', 'Item2', 'Item3', 'Item1']
```

[].fill()

`[].fill(valor, inicio, final)` preenche a array com o valor, do início até o fim.

```
'[Item1', 'Item2', 'Item3', 'Item4'].fill('Banana');  
// ['Banana', 'Banana', 'Banana', 'Banana']
```

```
'[Item1', 'Item2', 'Item3', 'Item4'].fill('Banana', 2);  
// ['Item1', 'Item2', 'Banana', 'Banana']
```

```
'[Item1', 'Item2', 'Item3', 'Item4'].fill('Banana', 1, 3);  
// ['Item1', 'Banana', 'Banana', 'Item4']
```

[].fill()

`[].fill(valor, inicio, final)` preenche a array com o valor, do início até o fim.

```
'[Item1', 'Item2', 'Item3', 'Item4'].fill('Banana');  
// ['Banana', 'Banana', 'Banana', 'Banana']
```

```
'[Item1', 'Item2', 'Item3', 'Item4'].fill('Banana', 2);  
// ['Item1', 'Item2', 'Banana', 'Banana']
```

```
'[Item1', 'Item2', 'Item3', 'Item4'].fill('Banana', 1, 3);  
// ['Item1', 'Banana', 'Banana', 'Item4']
```

Métodos de Acesso [].concat()

Os métodos abaixo não modificam a array original, apenas retornam uma array modificada. `[].concat()` irá concatenar a array com o valor passado.

```
const transporte1 = ['Barco', 'Aviao'];
const transporte2 = ['Carro', 'Moto'];
const transportes = transporte1.concat(transporte2);
// ['Barco', 'Aviao', 'Carro', 'Moto'];

const maisTransportes = [].concat(transporte1, transporte2,
'Van');
// ['Barco', 'Aviao', 'Carro', 'Moto', 'Van'];
```

[] .includes(), [] .indexOf() e [] .lastIndexOf()

`[] .includes(valor)` verifica se a array possui o valor e retorna true ou false. `[] .indexOf(valor)` verifica se a array possui o valor e retorna o index do primeiro valor na array. Já o `[] .lastIndexOf(valor)` retorna o index do último.

```
const linguagens = ['html', 'css', 'js', 'php', 'python',
'js'];

linguagens.includes('css'); // true
linguagens.includes('ruby'); // false
linguagens.indexOf('python'); // 4
linguagens.indexOf('js'); // 2
linguagens.lastIndexOf('js'); // 5
```

[] .join()

[] .join(separador) junta todos os valores da array e retorna uma string com eles. Se você passar um valor como parâmetro, este será utilizado durante a junção de cada item da array.

```
const linguagens = ['html', 'css', 'js', 'php', 'python'];
linguagens.join(); // 'html,css,js,php,python'
linguagens.join(' '); // 'html css js php python'
linguagens.join('-_-'); // 'html_-_-css_-_-js_-_-php_-_-python'

let htmlString = '<h2>Título Principal</h2>';
htmlString = htmlString.split('h2');
// [<, >Título Principal</, >]
htmlString = htmlString.join('h1');
// <h1>Título Principal</h1>
```

[].slice()

`[].slice(inicio, final)` retorna os itens da array começando pelo início e indo até o valor de final.

```
const linguagens = ['html', 'css', 'js', 'php', 'python'];
linguagens.slice(3); // ['php', 'python']
linguagens.slice(1, 4); // ['css', 'js', 'php']

const cloneLinguagens = linguagens.slice();
```

Exercícios

```
const comidas = ['Pizza', 'Frango', 'Carne', 'Macarrão'];
// Remova o primeiro valor de comidas e coloque em uma variável
// Remova o último valor de comidas e coloque em uma variável
// Adicione 'Arroz' ao final da array
// Adicione 'Peixe' e 'Batata' ao início da array

const estudantes = ['Marcio', 'Brenda', 'Joana', 'Kleber',
'Julia'];
// Arrume os estudantes em ordem alfabética
// Inverta a ordem dos estudantes
// Verifique se Joana faz parte dos estudantes
// Verifique se Juliana faz parte dos estudantes

let html = `<section>
    <div>Sobre</div>
    <div>Produtos</div>
    <div>Contato</div>
</section>`
// Substitua section por ul e div com li,
// utilizando split e join
```

```
// Remova o último carro, mas antes de remover  
// salve a array original em outra variável
```

OBJETOS

Array Iteração

[].forEach()

[].forEach(callback(itemAtual, index, array)) a função de callback é executada para cada item da array. Ela possui três argumentos, itemAtual (valor do item da array), index (index do valor na array) e array (array original).

```
const carros = ['Ford', 'Fiat', 'Honda'];
carros.forEach(function(item, index, array) {
    console.log(item.toUpperCase());
});

// com Arrow Function
carros.forEach((item, index, array) => {
    console.log(item.toUpperCase());
});
```

O método sempre retorna
undefined

Arrow Function

```
const li = document.querySelectorAll('li');

li.forEach(i => i.classList.add('ativa'));

li.forEach(function(item) {
  item.classList.add('ativa');
});
```

Modificar a Array Original

O terceiro argumento do callback é uma referência direta e se modificado irá também modificar a array original.

```
const carros = ['Ford', 'Fiat', 'Honda'];
carros.forEach((item, index, array) => {
  array[index] = 'Carro ' + item;
});

carros // ['Carro Ford', 'Carro Fiat', 'Carro Honda']
```

É melhor utilizarmos o map para isso

[].map()

[].map(callback(itemAtual, index, array)) funciona da mesma forma que o forEach(), porém ao invés de retornar undefined, retorna uma nova array com valores atualizados de acordo com o return de cada iteração.

```
const carros = ['Ford', 'Fiat', 'Honda'];
const newCarros = carros.map((item) => {
  return 'Carro ' + item;
});

carros; // ['Ford', 'Fiat', 'Honda']
newCarros; // ['Carro Ford', 'Carro Fiat', 'Carro Honda'];
```

Valor Retornado

Se não retornarmos nenhum valor durante a iteração utilizando map, o valor retornado como de qualquer função que não possui o return, será undefined.

```
const carros = ['Ford', 'Fiat', 'Honda'];
const newCarros = carros.map((item) => {
  const novoValor = 'Carro ' + item;
});

newCarros; // [undefined, undefined, undefined];
```

Arrow Function e [].map()

Uma Arrow Function de linha única e sem chaves irá retornar o valor após a fat arrow `=>`.

```
const numeros = [2, 4, 6, 8, 10, 12, 14];
const numerosX3 = numeros.map(n => n * 3);

numerosX3; // [6, 12, 18, 24, 30, 36, 42];
```

[].map() vs [].forEach()

Se o objetivo for modificar os valores da array atual, sempre utilize o map, pois assim uma nova array com os valores modificados é retornada e você pode imediatamente iterar novamente sobre estes valores.

```
const numeros = [2, 4, 6, 8, 10, 12, 14];
const numerosX3 = numeros.map(n => n * 3);

numerosX3; // [6, 12, 18, 24, 30, 36, 42];
```

[].map() com Objetos

Map pode ser muito útil para interagirmos com uma array de objetos, onde desejamos isolar um valor único de cada objeto.

```
const aulas = [  
  {  
    nome: 'HTML 1',  
    min: 15  
  },  
  {  
    nome: 'HTML 2',  
    min: 10  
  },  
  {  
    nome: 'CSS 1',  
    min: 20  
  },  
  {  
    nome: 'JS 1',  
    min: 25  
  },  
]
```

```
// [15, 10, 20, 25];

const puxarNomes = aula => aula.nome;
const nomesAulas = aulas.map(puxarNomes);
// ['HTML 1', 'HTML 2', 'CSS 1', 'JS 1']
```

[].reduce()

```
[ ].reduce(callback(acumulador, valorAtual, index,  
array), valorInicial)
```

executa a função de callback para cada item da Array. Um valor especial existe nessa função de callback, ele é chamado de **acumulador**, mas é na verdade apenas o retorno da iteração anterior.

```
const aulas = [10, 25, 30];  
const total1 = aulas.reduce((acumulador, atual) => {  
    return acumulador + atual;  
});  
total1; // 65  
  
const total2 = aulas.reduce((acc, cur) => acc + cur, 100);  
total2; // 165
```

Reduce Passo a Passo 1

O primeiro parâmetro do callback é o valor do segundo argumento passado no `reduce(callback, inicial)` durante a primeira iteração. Nas iterações seguintes este valor passa a ser o retornado pela anterior.

```
const aulas = [10, 25, 30];

// 1
aulas.reduce((0, 10) => {
  return 0 + 10;
}, 0); // retorna 10

// 2
aulas.reduce((10, 25) => {
  return 10 + 25;
}, 0); // retorna 35

// 3
aulas.reduce((35, 30) => {
  return 35 + 30;
}, 0); // retorna 65
```

ORIGAMID

Reduce Passo a Passo 2

Se não definirmos o valor inicial do acumulador, ele irá **pular** a primeira iteração e começará a partir da segunda. Neste caso o valor do acumulador será o valor do item da primeira iteração.

```
const aulas = [10, 25, 30];

// 1
aulas.reduce((10, 25) => {
  return 10 + 25;
}) // retorna 35

// 2
aulas.reduce((35, 30) => {
  return 35 + 30;
}) // retorna 65
```

Maior Valor com [].reduce()

```
const numeros = [10, 25, 60, 5, 35, 10];

const maiorValor = numeros.reduce((anterior, atual) => {
    return anterior < atual ? atual : anterior;
});

maiorValor; // 60
```

Podemos retornar outros valores

```
const aulas = [  
  {  
    nome: 'HTML 1',  
    min: 15  
  },  
  {  
    nome: 'HTML 2',  
    min: 10  
  },  
  {  
    nome: 'CSS 1',  
    min: 20  
  },  
  {  
    nome: 'JS 1',  
    min: 25  
  },  
]
```

```
const listaAulas = aulas.reduce((acumulador, atual, index) => {
```

{ , { }

Passo a passo Reduce

Passo a passo do método reduce criando um Objeto.

```
// 1
aulas.reduce(({}, {nome: 'HTML 1', min: 15}, 0) => {
  {}[0] = 'HTML 1';
  return {0: 'HTML 1'};
}, {})

// 2
aulas.reduce({0: 'HTML 1'}, {nome: 'HTML 2', min: 10}, 1) => {
  {0: 'HTML 1'}[1] = 'HTML 2';
  return {0: 'HTML 1', 1: 'HTML 2'};
}, {})

// 3
aulas.reduce({0: 'HTML 1', 1: 'HTML 2'}, {nome: 'CSS 1', min: 20}, 2) => {
  {0: 'HTML 1', 1: 'HTML 2'}[2] = 'CSS 1';
  return {0: 'HTML 1', 1: 'HTML 2', 2: 'CSS 1'};
}, {})
```

```
aulas.reduce(({0: 'HTML 1', 1: 'HTML 2', 2: 'CSS 1'}, {nome:  
'JS 1', min: 25}, 3) => {  
  {0: 'HTML 1', 1: 'HTML 2', 2: 'CSS 1'}[3] = 'JS 1';  
  return {0: 'HTML 1', 1: 'HTML 2', 2: 'CSS 1', 3: 'JS 1'};  
}, {})
```

[].reduceRight()

Existe também o método `[].reduceRight()`, a diferença é que este começa a iterar da direita para a esquerda, enquanto o reduce itera da esquerda para a direita.

```
const frutas = ['Banana', 'Pêra', 'Uva'];

const frutasRight = frutas.reduceRight((acc, fruta) => acc + ' ' + fruta);
const frutasLeft = frutas.reduce((acc, fruta) => acc + ' ' + fruta);

frutasRight; // Uva Pêra Banana
frutasLeft; // Banana Pêra Uva
```

[].some()

[].some(), se pelo menos um return da iteração for truthy, ele retorna true.

```
const frutas = ['Banana', 'Pêra', 'Uva'];
const temUva = frutas.some((fruta) => {
  return fruta === 'Uva';
}); // true

function maiorQue100(numero) {
  return numero > 100;
}
const numeros = [0, 43, 22, 88, 101, 2];
const temMaior = numeros.some(maiorQue100); // true
```

[].every()

[].every() , se todos os returns das iterações forem truthy, o método irá retornar true. Se pelo menos um for falsy, ele irá retornar false.

```
const frutas = ['Banana', 'Pêra', 'Uva', ''];
// False pois pelo menos uma fruta
// está vazia '', o que é um valor falsy
const arraysCheias = frutas.every((fruta) => {
    return fruta; // false
});

const numeros = [6, 43, 22, 88, 101, 29];
const maiorQue3 = numeros.every(x => x > 3); // true
```

[] .find() e [] .findIndex()

`[] .find()`, retorna o valor atual da primeira iteração que retornar um valor truthy. Já o `[] .findIndex()`, ao invés de retornar o valor, retorna o index deste valor na array.

```
const frutas = ['Banana', 'Pêra', 'Uva', 'Maçã'];
const buscaUva = frutas.findIndex((fruta) => {
  return fruta === 'Uva';
}); // 2

const numeros = [6, 43, 22, 88, 101, 29];
const buscaMaior45 = numeros.find(x => x > 45); // 88
```

[].filter()

`[].filter()`, retorna uma array com a lista de valores que durante a sua iteração retornaram um valor truthy.

```
const frutas = ['Banana', undefined, null, '', 'Uva', 0,  
'Maçã'];  
const arrayLimpa = frutas.filter((fruta) => {  
    return fruta;  
}); // ['Banana', 'Uva', 'Maçã']  
  
const numeros = [6, 43, 22, 88, 101, 29];  
const buscaMaior45 = numeros.filter(x => x > 45); // [88, 101]
```

Filter em Objetos

```
const aulas = [  
  {  
    nome: 'HTML 1',  
    min: 15  
  },  
  {  
    nome: 'HTML 2',  
    min: 10  
  },  
  {  
    nome: 'CSS 1',  
    min: 20  
  },  
  {  
    nome: 'JS 1',  
    min: 25  
  },  
]
```

```
const aulasMaiores = aulas.filter((aula) => {
```

```
// [{nome: 'CSS 1', min: 20}, {nome: 'JS 1', min: 25}]
```

Exercícios

```
<section class="curso">
  <h1>Web Design Completo</h1>
  <p>Este curso é para quem deseja entrar ou já está no mercado de criação de websites.</p>
  <span class="aulas">80</span>
  <span class="horas">22</span>
</section>
<section class="curso">
  <h1>WordPress Como CMS</h1>
  <p>No curso de WordPress Como CMS, você aprende do zero como pegar qualquer site em HTML e torná-lo totalmente gerenciável com a plataforma do WordPress.</p>
  <span class="aulas">46</span>
  <span class="horas">9</span>
</section>
<section class="curso">
  <h1>UI Design Avançado</h1>
  <p>Este é um curso avançado de User Interface Design.</p>
  <span class="aulas">55</span>
```

```
// Seleciona cada curso e retorne uma array
// com objetos contendo o título, descrição,
// aulas e horas de cada curso

// Retorne uma lista com os
// números maiores que 100
const numeros = [3, 44, 333, 23, 122, 322, 33];

// Verifique se Baixo faz parte
// da lista de instrumentos e retorne true
const instrumentos = ['Guitarra', 'Baixo', 'Bateria',
'Teclado']

// Retorne o valor total das compras
const compras = [
{
    item: 'Banana',
    preço: 'R$ 4,99'
```

```
        item: 'Ovo',
        preco: 'R$ 2,99'
    },
{
    item: 'Carne',
    preco: 'R$ 25,49'
},
{
    item: 'Refrigerante',
    preco: 'R$ 5,35'
},
{
    item: 'Queijo',
    preco: 'R$ 10,60'
}
]
```

OBJETOS

Function

Function

Toda função é criada com o construtor Function e por isso herda as suas propriedades e métodos.

```
function areaQuadrado(lado) {  
    return lado * lado;  
}  
  
const perimetroQuadrado = new Function('lado', 'return lado *  
4');
```

Propriedades

`Function.length` retorna o total de argumentos da função.

`Function.name` retorna uma string com o nome da função.

```
function somar(n1, n2) {  
    return n1 + n2;  
}
```

```
somar.length; // 2  
somar.name; // 'somar'
```

function.call()

`function.call(this, arg1, arg2, ...)` executa a função, sendo possível passarmos uma nova referência ao `this` da mesma.

```
const carro = {  
    marca: 'Ford',  
    ano: 2018  
}  
  
function descricaoCarro() {  
    console.log(this.marca + ' ' + this.ano);  
}  
  
descricaoCarro() // undefined undefined  
descricaoCarro.call() // undefined undefined  
descricaoCarro.call(carro) // Ford 2018
```

This

O valor de this faz referência ao objeto criado durante a construção do objeto (Constructor Function). Podemos trocar a referência do método ao this, utilizando o `call()`.

```
const carros = ['Ford', 'Fiat', 'VW'];

carros.forEach((item) => {
  console.log(item);
}); // Log de cada Carro

carros.forEach.call(carros, (item) => {
  console.log(item);
}); // Log de cada Carro

const frutas = ['Banana', 'Pêra', 'Uva'];

carros.forEach.call(frutas, (item) => {
  console.log(item);
}); // Log de cada Fruta
```

Exemplo Real

O objeto atribuído a `lista` será o substituído pelo primeiro argumento de `call()`

```
function Dom(seletor) {  
    this.element = document.querySelector(seletor);  
}  
  
Dom.prototype.ativo = function(classe) {  
    this.element.classList.add(classe);  
}  
  
const lista = new Dom('ul');  
lista.ativo('ativar');  
console.log(lista);
```

O Objeto deve ser parecido

O novo valor de this deve ser semelhante a estrutura do valor do this original do método. Caso contrário o método não conseguirá interagir de forma correta com o novo this.

```
const novoSeletor = {  
    element: document.querySelector('li')  
}  
  
Dom.prototype.ativo.call(novoSeletor, 'ativar');
```

Array's e Call

É comum utilizarmos o `call()` nas funções do protótipo do construtor Array. Assim podemos estender todos os métodos de Array à objetos que se parecem com uma Array (array-like).

```
Array.prototype.mostreThis = function() {  
    console.log(this);  
}  
  
const frutas = ['Uva', 'Maçã', 'Banana'];  
frutas.mostreThis(); // ['Uva', 'Maçã', 'Banana']  
  
Array.prototype.pop.call(frutas); // Remove Banana  
frutas.pop(); // Mesma coisa que a função acima
```

Array-like

HTMLCollection, NodeList e demais objetos do Dom, são parecidos com uma array. Por isso conseguimos utilizar os mesmos na substituição do this em call.

```
const li = document.querySelectorAll('li');

const filtro = Array.prototype.filter.call(li, function(item) {
    return item.classList.contains('ativo');
});

filtro; // Retorna os itens que possuem ativo
```

function.apply()

O `apply(this, [arg1, arg2, ...])` funciona como o call, a única diferença é que os argumentos da função são passados através de uma array.

```
const numeros = [3,4,6,1,34,44,32];
Math.max.apply(null, numeros);
Math.max.call(null, 3, 4, 5, 6, 7, 20);
```

```
// Podemos passar null para o valor
// de this, caso a função não utilize
// o objeto principal para funcionar
```

Apply vs Call

A única diferença é a array como segundo argumento.

```
const li = document.querySelectorAll('li');

function itemPossuiAtivo(item) {
    return item.classList.contains('ativo');
}

const filtro1 = Array.prototype.filter.call(li,
itemPossuiAtivo);
const filtro2 = Array.prototype.filter.apply(li,
[itemPossuiAtivo]);
```

function.bind()

Diferente de call e apply, `bind(this, arg1, arg2, ...)` não irá executar a função mas sim retornar a mesma com o novo contexto de `this`.

```
const li = document.querySelectorAll('li');

const filtrarLi = Array.prototype.filter.bind(li,
function(item) {
    return item.classList.contains('ativo');
});

filtrarLi();
```

Argumentos e Bind

Não precisamos passar todos os argumentos no momento do bind, podemos passar os mesmos na nova função no momento da execução da mesma.

```
const carro = {  
    marca: 'Ford',  
    ano: 2018,  
    acelerar: function(aceleracao, tempo) {  
        return `${this.marca} acelerou ${aceleracao} em ${tempo}`;  
    }  
};  
carro.acelerar(100, 20);  
// Ford acelerou 100 em 20  
  
const honda = {  
    marca: 'Honda',  
};  
const acelerarHonda = carro.acelerar.bind(honda);  
acelerarHonda(200, 10);  
// Honda acelerou 200 em 10
```

ORIGAMID

Argumentos Comuns

Podemos passar argumentos padrões para uma função e retornar uma nova função.

```
function imc(altura, peso) {  
  return peso / (altura * altura);  
  
}  
  
const imc180 = imc.bind(null, 1.80);  
  
imc(1.80, 70); // 21.6  
imc180(70); // 21.6
```

Exercícios

```
<section>
```

```
    <p>Lobo-cinzento (nome científico:Canis lupus) é uma espécie  
de mamífero canídeo do gênero Canis. É um sobrevivente da Era  
do Gelo, originário do Pleistoceno Superior, cerca de 300 mil  
anos atrás. É o maior membro remanescente selvagem da família  
canidae.</p>
```

```
    <p>Os lobos-cinzentos são tipicamente predadores ápice nos  
ecossistemas que ocupam. Embora não sejam tão adaptáveis à  
presença humana como geralmente ocorre com as demais.</p>
```

```
    <p>O peso e tamanho dos lobos variam muito em todo o mundo,  
tendendo a aumentar proporcionalmente com a latitude.</p>
```

```
    <p>Os lobos são capazes de percorrer longas distâncias com  
uma velocidade média de 10 quilômetros por hora e são  
conhecidos por.</p>
```

```
</section>
```

```
// Retorne a soma total de caracteres dos  
// parágrafos acima utilizando reduce
```

```
// html, com os seguintes parâmetros  
// tag, classe e conteúdo.
```

```
// Crie uma nova função utilizando a anterior como base  
// essa nova função deverá sempre criar h1 com a  
// classe título. Porém o parâmetro conteúdo continuará  
dinâmico
```

OBJETOS

Object

Object

Todo objeto é criado com o construtor Object e por isso herda as propriedades e métodos do seu prototype.

```
const carro = {  
    marca: 'Ford',  
    ano: 2018,  
}  
  
const pessoa = new Object({  
    nome: 'André',  
    idade: 28,  
})
```

Métodos de Object

`Object.create(obj, defineProperties)` retorna um novo objeto que terá como protótipo o objeto do primeiro argumento.

```
const carro = {
    rodas: 4,
    init(marca) {
        this.marca = marca;
        return this;
    },
    acelerar() {
        return `${this.marca} acelerou as ${this.rodas} rodas`;
    },
    buzinar() {
        return this.marca + ' buzinou';
    }
}

const honda = Object.create(carro);
honda.init('Honda').acelerar();
```

Object.assign()

`Object.assign(alvo, obj1, obj2)` adiciona ao alvo as propriedades e métodos enumeráveis dos demais objetos. O assign irá modificar o objeto alvo.

```
const funcaoAutomovel = {  
    acelerar() {  
        return 'acelerou';  
    },  
    buzinar() {  
        return 'buzinou';  
    },  
}
```

```
const moto = {  
    rodas: 2,  
    capacete: true,  
}
```

```
const carro = {  
    rodas: 4,  
    mala: true,
```

```
Object.assign(moto, funcaoAutomovel);  
Object.assign(carro, funcaoAutomovel);
```

Object.defineProperties()

`Object.defineProperties(alvo, propriedades)` adiciona ao alvo novas propriedades. A diferença aqui é a possibilidade de serem definidas as características dessas propriedades.

```
const moto = {};
Object.defineProperties(moto, {
    rodas: {
        value: 2,
        configurable: false, // impede deletar e mudança de valor
        enumerable: true, // torna enumerável
    },
    capacete: {
        value: true,
        configurable: true,
        writable: false, // impede mudança de valor
    },
})

moto.rodas = 4;
delete moto.capacete;
moto;
```

*Existe também o
Object.defineProperty, para uma
propriedade única.*

get e set

É possível definirmos diferentes comportamentos para get e set de uma propriedade. Lembrando que ao acionarmos uma propriedade `obj.propriedade`, a função get é ativada e ao setarmos `ob.propriedade = 'Valor'` a função de set é ativada.

```
const moto = {};
Object.defineProperties(moto, {
  velocidade: {
    get() {
      return this._velocidade;
    },
    set(valor) {
      this._velocidade = 'Velocidade ' + valor;
    }
  }
});

moto.velocidade = 200;
moto.velocidade;
// Velocidade 200
```

ORIGAMID

Object.getOwnPropertyDescriptors(obj)

Lista todos os métodos e propriedades de um objeto, com as suas devidas configurações.

```
Object.getOwnPropertyDescriptors(Array);  
// Lista com métodos e propriedades do Array
```

```
Object.getOwnPropertyDescriptors(Array.prototype);  
// Lista com métodos e propriedades do protótipo de Array
```

```
Object.getOwnPropertyDescriptor(window, 'innerHeight');  
// Puxa de uma única propriedade
```

Object.keys(obj), Object.values(obj) Object.entries(obj)

`Object.keys(obj)` retorna uma array com as chaves de todas as propriedades diretas e enumeráveis do objeto.

`Object.values(obj)` retorna uma array com os valores do objeto. `Object.entries(obj)` retorna uma array com array's contendo a chave e o valor.

```
Object.keys(Array);
// [] vazia, pois não possui propriedades enumeráveis

const carro = {
  marca: 'Ford',
  ano: 2018,
}

Object.keys(carro);
// ['marca', 'ano']
Object.values(carro);
// ['Ford', 2018]
Object.entries(carro);
// [['marca', 'Ford'], ['ano', 2018]]
```

Object.getOwnPropertyNames(obj)

Retorna uma array com todas as propriedades diretas do objeto (não retorna as do protótipo).

```
Object.getOwnPropertyNames(Array);
// ['length', 'name', 'prototype', 'isArray', 'from', 'of']

Object.getOwnPropertyNames(Array.prototype);
// [..., 'filter', 'map', 'every', 'some', 'reduce', ...]

const carro = {
  marca: 'Ford',
  ano: 2018,
}
Object.getOwnPropertyNames(carro);
// ['marca', 'ano']
```

Object.getPrototypeOf() e Object.is()

`Object.getPrototypeOf()`, retorna o protótipo do objeto.
`Object.is(obj1, obj2)` verifica se os objetos são iguais e retorna true ou false.

```
const frutas = ['Banana', 'Pêra']
Object.getPrototypeOf(frutas);
Object.getPrototypeOf(''); // String

const frutas1 = ['Banana', 'Pêra'];
const frutas2 = ['Banana', 'Pêra'];

Object.is(frutas1, frutas2); // false
```

Object.freeze(), Object.seal(), Object.preventExtensions()

`Object.freeze()` impede qualquer mudança nas propriedades.

`Object.seal()` previne a adição de novas propriedades e impede que as atuais sejam deletadas.

`Object.preventExtensions()` previne a adição de novas propriedades.

```
const carro = {  
    marca: 'Ford',  
    ano: 2018,  
}  
Object.freeze(carro);  
Object.seal(carro);  
Object.preventExtensions(carro);  
  
Object.isFrozen(carro); // true  
Object.isSealed(carro); // true  
Object.isExtensible(carro); // true
```

Propriedades e Métodos do Protótipo

Já que tudo em JavaScript é objeto, as propriedades abaixo estão disponíveis em todos os objetos criados a partir de funções construtoras. `{}.constructor` retorna a função construtora do objeto.

```
const frutas = ['Banana', 'Uva'];
frutas.constructor; // Array
```

```
const frase = 'Isso é uma String';
frase.constructor; // String
```

`{}.hasOwnProperty('prop')` e
`{}.propertyIsEnumerable('prop')`

Verifica se possui a propriedade e retorna true. A propriedade deve ser direta do objeto e não do protótipo. O

`{}.propertyIsEnumerable()` verifica se a propriedade é enumerável.

```
const frutas = ['Banana', 'Uva'];

frutas.hasOwnProperty('map'); // false
Array.hasOwnProperty('map'); // false
Array.prototype.hasOwnProperty('map'); // true

Array.prototype.propertyIsEnumerable('map'); // false
window.propertyIsEnumerable('innerHeight'); // true
```

{}.isPrototypeOf(valor)

Verifica se é o protótipo do valor passado.

```
const frutas = ['Banana', 'Uva'];

Array.prototype.isPrototypeOf(frutas); // true
```

{}.toString()

Retorna o tipo do objeto. O problema é `toString()` ser uma função dos protótipos de `Array`, `String` e mais. Por isso é comum utilizarmos a função direto do

```
Object.prototype.toString.call(valor).
```

```
const frutas = ['Banana', 'Uva'];
frutas.toString(); // 'Banana,Uva'
typeof frutas; // object
Object.prototype.toString.call(frutas); // [object Array]
```

```
const frase = 'Uma String';
frase.toString(); // 'Uma String'
typeof frase; // string
Object.prototype.toString.call(frase); // [object String]
```

```
const carro = {marca: 'Ford'};
carro.toString(); // [object Object]
typeof carro; // object
Object.prototype.toString.call(carro); // [object Object]
```

```
const li = document.querySelectorAll('li');
```

```
Object.prototype.toString.call(li); // [object NodeList]
```

Exercícios

```
// Crie uma função que verifique  
// corretamente o tipo de dado
```

```
// Crie um objeto quadrado com  
// a propriedade lados e torne  
// ela imutável
```

```
// Previna qualquer mudança  
// no objeto abaixo  
const configuracao = {  
  width: 800,  
  height: 600,  
  background: '#333'  
}
```

```
// Liste o nome de todas  
// as propriedades do  
// protótipo de String e Array
```