

JavaScript

FUNCTION



Function

this

getter

setter

call

apply

bind

new

prototype

Function

Uma função é um objeto que contém um bloco de código executável

```
Function functionName(parameters) {  
    // function body // ...  
}
```

Frequentemente, precisamos realizar uma ação semelhante em muitos lugares do script

```
// CHAMANDO UMA FUNÇÃO
function showMessage() {
    alert( 'Hi Developers!' );
}
```

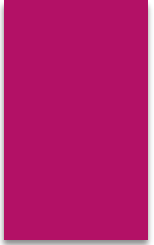
```
showMessage();
showMessage(); // podemos evocar uma função muitas vezes
```

```
// VARIÁVEIS EXTERNAS
let userName = 'Eduardo';

function showMessage() {
    let message = 'Oi, ' + userName;
    alert(message);
}
```

```
showMessage();
```

A função tem acesso total à variável externa. Ela também pode modificá-lo



```
// ALTERANDO OS VALORES VRIAVEL EXTERNA

let userName = 'Eduardo';

function showMessage() {
    userName = "Carlinhos"; // Alterando a variável externa
    let message = 'Oi, ' + userName; alert(message);
}

alert(userName); // Eduardo é mostrado
showMessage(); // Carlinhos é mostrado
alert(userName); // Carlinhos é mostrado novamente
```

A variável externa só é usada se não houver uma local

Valores padrão

```
function showMessage(from, text = "nenhum texto fornecido") {  
    alert( from + ": " + text );  
}  
showMessage("Ana"); // Ana: nenhum texto fornecido  
// "nenhum texto fornecido" é uma string, pode ser uma expressão  
complexa, que só é avaliada e atribuída se o parâmetro estiver ausente
```

Parâmetros padrão alternativos

```
function showMessage(text) {  
    // ...  
    if (text === undefined) { // se o parâmetro estiver faltando  
        text = 'mensagem vazia';  
    }  
    alert(text);  
}  
showMessage(); // mensagem vazia
```

Function Parâmetros

Cada função em JavaScript **retorna undefined**, a menos que seja especificada outra forma

```
function say(message) {  
    console.log(message);  
}  
let result = say('Olá');
```

```
console.log('Resultado:', result); // Apenas undefined  
console.log('Olá'); // mostra Olá mas continua sem definição
```

Para especificar um valor de retorno para uma função, use a **instrução return** seguida por uma expressão ou um valor

```
// PASSANDO PARÂMETROS COM RETURN
```

```
function sum(a, b) {  
    return a + b;  
}
```

```
sum; // Sem passar os parâmetros, o JS retorna o tipo [Function]
```

```
sum(2, 2); // passando parâmetros
```




```
// VALORES PADRÃO NA PASSAGEM DE PARAMETRO
```

```
function sum(a = 1, b = 1) {  
    return a + b;  
}
```

```
sum(2, 2);
```

```
sum(5);
```

```
sum();
```

É possível invocar uma função com menos ou mais parâmetros, não necessariamente seguindo o que está declarado

Function

Qual é a diferença entre **function declaration** e **expression**?

// FUNCTION DECLARATION

```
sum(2, 2);  
// inverter a chamada funciona  
function sum(a, b) {  
    return a + b;  
}
```

// FUNCTION EXPRESSION

```
sum(3, 3); // NÃO funciona!  
const sum = function(x, y) {  
    return x + y;  
}
```

Function

As funções em JS
são de primeira classe, ou seja, podem ser
atribuídas a uma variável, passadas por parâmetro
ou serem retornada de uma outra função

// FUNÇÃO QUE CHAMA FUNÇÃO

```
let soma = function(a, b) {  
    return a + b;  
};
```

```
let subtrai = function(a, b) {  
    return a - b;  
};
```

```
let calculator = function(fn) {  
    return function(a, b) { // essa função retorna outra função  
        return fn(a, b);  
    }  
}; // atente pelos pontos e virgulas
```

```
console.log(calculator(soma)(2, 2));  
console.log(calculator(subtrai)(2, 2));
```

Function arguments

Por meio da variável implícita **arguments** é possível acessar os parâmetros da função invocada

```
// ARGUMENTS
const sum = function() {
  console.log(arguments);
}
sum(1,2,3,4,5,6,7,8,9);
```

```
// arguments não tem as
propriedades do array
```

```
let sum = function() {
  let total = 0;
  for(let argument in arguments) {
    total += arguments[argument];
  }
  return total;
};
sum(1,2,3,4,5,6,7,8,9);
// mude argument para banana
```

Function rest parameter

Também é possível acessar os parâmetros da função invocada por meio do **rest parameter** definido pelos: ...

```
let sum = function(...meusnumeros) {  
  let total = 0;  
  for(let numeros of meusnumeros) { //A diferença está no OFF (Array)  
    total += numeros; // Aqui soma-se os itens do array  
  }  
  return total;  
};  
console.log(sum(1,2,3,4,5,6,7,8,9));
```

Introduzido no ES6. Funciona como um array.

O **rest parameter** deve ser sempre o último da lista de parâmetros

```
// REST sempre como último da lista

let sum = function(a, b, c, ...numbers) {
  let total = a + b + c;
  //console.log(numbers); // veja as posições do rest
  for(let number of numbers) {
    total += number;
  }
  return total;
};

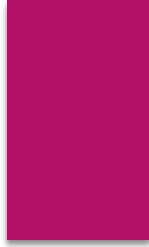
console.log(sum(1,2,3,4,5,6,7,8,9));
//inverta para gerar o erro
```

Introduzido no ES6. Funciona como um array.

this

Existe uma **variável implícita** chamada de **this** que faz referência para o objeto responsável pela sua invocação

```
// THIS - variavel implicita
const rectangle = {
  x: 10,
  y: 2,
  calculateArea: function() { // crie um método para calcular
    return this.x * this.y; // se rodar apenas com x e Y - ERRO
  }
};
console.log(rectangle.calculateArea());
// this se refere a rectangle nesse caso
```

```
// SIMPLIFICANDO COM ESSA SINTAXE (ES6)

const rectangle = {
  x: 10,
  y: 2,
  calculateArea() { // ISSO É UMA FUNÇÃO
                      (method notation)

    return this.x * this.y;
  }
};

console.log(rectangle.calculateArea());

// o interpretador entende que é uma função
```



```
// Passando calculateArea por REFERENCIA
```

```
const calculateArea = function() {  
    return this.x * this.y;  
};
```

```
const rectangle = {  
    x: 10,  
    y: 2,  
    calculateArea  
};
```

```
console.log(rectangle.calculateArea());
```

```
// o THIS se refere ao objeto que  
// está evocando a função(rectangle)
```

getter e setter

As funções do tipo **getter** e **setter** servem para interceptar o acesso as propriedades de determinado um objeto

```
// getter
const rectangle = {
  x: 10,
  y: 2,
  get area() { //method notation
    return this.x * this.y; }
};
console.log(rectangle.area);
// mais simples
```



```
// setter - VEJA O ERRO + COMUM
```

```
const rectangle = {  
  set x(x) {  
    this.x = x;  
  },  
  set y(y) {  
    this.y = y;  
  },  
  get area() {  
    return this.x * this.y;  
  }  
};  
rectangle.x = 10;  
rectangle.y = 2;  
console.log(rectangle.area);
```

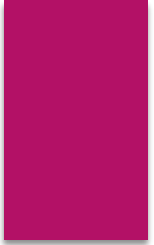
```
//esse é um erro muito comum
```

```
// set e this são executados em loop (são o  
mesmo)
```

Utilize chaves diferentes para a função setter e a propriedade do objeto

```
// MUDANDO O NOME DAS CHAVES
const rectangle = {
  set x(x) {
    this._x = x; // INTERNAMENTE É _x _y
  },
  set y(y) {
    this._y = y;
  },
  get area() {
    return this._x * this._y;
  }
};

rectangle.x = 10; // AQUI PERMANECE
rectangle.y = 2;
console.log(rectangle.area);
```



```
const rectangle = {
  set x(x) {
    if (x > 0) {
      this._x = x;
    } else {
      console.log("Invalid value for x");
    }
  },
  set y(y) {
    if (y > 0) {
      this._y = y;
    } else {
      console.log("Invalid value for y");
    }
  },
  get area() {
    return this._x * this._y;
  }
};
```

```
rectangle.x = -10;
rectangle.y = -2;
console.log(rectangle.area);
// voltando a valores positivos volta a funcionar
```

getter e setter - defineProperty

Por meio da operação `defineProperty` da Object API, também é possível definir funções do tipo getter e setter

```
const rectangle = {};  
Object.defineProperty(rectangle, "area", {  
  get() {  
    return this.x * this.y;  
  }  
});  
rectangle.x = 10;  
rectangle.y = 2;  
console.log(rectangle.area);
```

```
// defineProperties  
// antes do ES6 era assim  
// utilize o method notation - get and set  
  
// Recebe 3 propriedades  
// 1 - objeto alvo (rectangle)  
// 2 - nome da propriedade ("area")  
// 3 - Atributo get
```

call, apply e bind

Por meio das operações **call** e **apply** é possível invocar uma função passando o **this** por parâmetro

```
// agora sim utilizando o call e passando
// o objeto circle como parametro

const calculateArea = function() {
    return Math.PI * Math.pow(this.radius, 2);
};

const circle = {
    radius: 10,
    calculateArea
};

console.log(calculateArea.call(circle));
```




```
// nesse caso o apply funciona igualzinho ao call
```

```
const calculateArea = function() {  
    return Math.PI * Math.pow(this.radius, 2);  
};
```

```
const circle = {  
    radius: 10,  
    calculateArea  
};
```

```
console.log(calculateArea.apply(circle));
```

Qual é a diferença entre **call** e **apply** ?

```
// podemos passar uma função de arredondamento para o PI - fn
// e no parametro - Math.round ou Math.ceil
```

```
const calculateArea = function(fn) {
  return fn(Math.PI * Math.pow(this.radius, 2));
};
const circle = {
  radius: 10,
  calculateArea
};
console.log(calculateArea.call(circle, Math.round));
// o 1 parametro sempre o this
// do 2 pra frente, entram como parâmetro da função

console.log(calculateArea.apply(circle, [Math.ceil]));
// APPLY SEMPRE EM FORMA DE ARRAY
// remova o array e veja o erro
```

Veremos o **apply** em **new** mais adiante

A operação **bind** permite encapsular o **this** dentro da função, retornando-a

```
// bind
const calculateArea = function(fn) {
  return fn(Math.PI * Math.pow(this.radius, 2));
};

const circle = {
  radius: 10,
  calculateArea
};

const calculateAreaForCircle = calculateArea.bind(circle);
// encapsulando o this antes mesmo da execução


console.log(calculateAreaForCircle(Math.round));
console.log(calculateAreaForCircle(Math.ceil));
```



new

Como fazer para criar um objeto a partir da mesma estrutura?

A função **fábrica**, que é um tipo de padrão, retorna um novo objeto após ser invocada diretamente



```
// Criando 1 pessoa
const person = {
  name: "Linus Torvald",
  city: "Helsinki",
  year: 1969,
  getAge() {
    return ((new Date()).getFullYear() - this.year);
  }
};

// MÉTODO PARA CALCULAR A IDADE

console.log(person);
console.log(person.getAge());
```

```
// VAMOS CRIAR MAIS UMA PESSOA
```

```
const person1 = {  
  name: "Linus Torvald",  
  city: "Helsinki",  
  year: 1969,  
  getAge() {  
    return ((new Date()).getFullYear() - this.year);  
  }  
};
```

```
const person2 = {  
  name: "Bill Gates",  
  city: "Seattle",  
  year: 1955,  
  getAge() {  
    return ((new Date()).getFullYear() - this.year);  
  }  
};
```

```
console.log(person1);  
console.log(person1.getAge());  
console.log(person2);  
console.log(person2.getAge());
```

```
// REPARE QUE TODA ESTRUTURA DOS OBJETOS SÃO IGUAIS
```

// FUNÇÃO FÁBRICA

```
const createPerson = function(name, city, year) {  
  return {  
    name,  
    city,  
    year,  
    getAge() {  
      return ((new Date()).getFullYear() - this.year);  
    }  
  }  
};  
  
const person1 = createPerson("Linus Torvald", "Helsinki", 1969);  
const person2 = createPerson("Bill Gates", "Seattle", 1955);  
  
console.log(person1);  
console.log(person1.getAge());  
console.log(person2);  
console.log(person2.getAge());  
  
// Repare nos resultados que GET AGE está duplicado
```



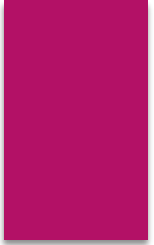
O que fazer para **eliminar a duplicação e reusar propriedades** entre os objetos?

A **função construtora** retorna um novo objeto ao ser invocada por meio do operador new


```
// FUNÇÃO CONSTRUTORA
// COLOQUE SEMPRE EM MAISUCULA (por convenção)
const Person = function(name, city, year) {
  this.name = name,
  this.city = city,
  this.year = year,
  this.getAge = function() {
    return ((new Date()).getFullYear() - this.year);
  }
};

const person1 = new Person("Linus Torvald", "Helsinki", 1969);
const person2 = new Person("Bill Gates", "Seattle", 1955);

console.log(person1);
console.log(person1.__proto__);
console.log(person1.getAge());
console.log(person2);
console.log(person2.__proto__);
console.log(person2.getAge());
console.log(person1.__proto__ === person2.__proto__);
// repare que elas são idênticas, reuso de código
// Repare nos resultados que GET AGE ainda está duplicado
```



Toda função **tem uma propriedade chamada prototype**, que é vinculada ao `__proto__` do objeto criado pelo operador `new`

Esse **prototype** é diferente do `__proto__`, apenas funções possuem. Porém é vinculada ao `__proto__`

```
// PROTOTYPE
// Apenas as funções construtoras usam o prototype
const Person = function(name, city, year) {
    this.name = name,
    this.city = city,
    this.year = year
};
Person.prototype.getAge = function() {
    return ((new Date()).getFullYear() - this.year);
};
//Colocando todas as propriedades comuns
// na função construtora (person) .prototype
//todos os objetos passam a compartilhar
const person1 = new Person("Linus Torvald", "Helsinki", 1969);
const person2 = new Person("Bill Gates", "Seattle", 1955);

console.log(person1);
console.log(person1.__proto__);
console.log(person1.getAge());
console.log(person2);
console.log(person2.__proto__);
console.log(person2.getAge());
console.log(person1.__proto__ === person2.__proto__);
// repare que elas são idênticas
// Repare nos resultados que GET AGE NÃO está duplicado
```