

JavaScript

ARROW FUNCTION



Arrow Function

InstanceOf

Tratamento de Exceções

Destructuring

JSON

Arrow functions

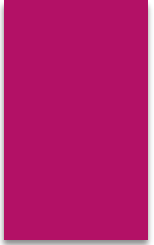
As **arrow functions** tem uma abordagem mais simples e direta para escrever uma função e podem melhorar a legibilidade do código em diversas situações



```
// arrow functions - Melhoram a legibilidade
// do código - Lembram desse exemplo de function?!
```

```
const sum = function(a, b) {
  return a + b;
};
const subtract = function(a, b) {
  return a - b;
};
const calculator = function(fn) {
  return function(a, b) {
    return fn(a, b);
  };
};

console.log(calculator(sum)(2, 2));
console.log(calculator(subtract)(2, 2));
```



```
// removemos a palavra function e trocamos pelo  
// simbolo de arrow
```

```
const sum = (a, b) => {    //aqui  
    return a + b;  
};  
const subtract = (a, b) => { //aqui  
    return a - b;  
};  
const calculator = (fn) => { //aqui  
    return (a, b) => {      //aqui  
        return fn(a, b);  
    };  
};
```

```
console.log(calculator(sum)(2, 2));  
console.log(calculator(subtract)(2, 2));  
// ficou mais enxuto, mas não necessariamente  
// mais legível
```

// vamos melhorar removendo as chaves e o return (return é AUTOMÁTICO)

```
const sum = (a, b) => a + b;  
const subtract = (a, b) => a - b;  
const calculator = (fn) => (a, b) => fn(a, b);  
// temos duas funções aninhadas aqui nem sempre é legível
```

```
console.log(calculator(sum)(2, 2));  
console.log(calculator(subtract)(2, 2));
```

// Com dois ou mais parâmetros o parênteses é obrigatório. Com um não, veja FN

```
const sum = (a, b) => a + b;  
const subtract = (a, b) => a - b;  
const calculator = fn => (a, b) => fn(a, b);
```

```
console.log(calculator(sum)(2, 2));  
console.log(calculator(subtract)(2, 2));
```

Arrow functions - ATENÇÃO

As arrow functions não possuem as suas próprias variáveis **this** e **arguments**

```
const person = {  
  name: "James Gosling",  
  city: "Alberta",  
  year: 1955,  
  getAge: function() {  
    return (new Date()).getFullYear() - this.year;  
  }  
};  
console.log(person);  
console.log(person.getAge());
```



```
// Não utilize arrow function com o método this
```

```
const person = {  
  name: "James Gosling",  
  city: "Alberta",  
  year: 1955,  
  getAge: () => {  
    return (new Date()).getFullYear() - this.year;  
  }  
};  
console.log(person);  
console.log(person.getAge());
```

```
// o this do objeto evocado(person) não funciona
```


// Da mesma maneira, o arguments não funciona
// vamos ver funcionando primeiro

```
const sum = function() {  
  let total = 0;  
  for(let argument in arguments) {  
    total += arguments[argument];  
  }  
  return total;  
};  
console.log(sum(1,2,3,4,5,6,7,8,9));
```

// Quando colocamos a arrow function com arguments não funciona

```
const sum = () => {  
  let total = 0;  
  for(let argument in arguments) {  
    total += arguments[argument];  
  }  
  return total;  
};  
console.log(sum(1,2,3,4,5,6,7,8,9));
```

E se um **objeto** for retornado?

```
const createPerson = function(name, city, year) {  
  return {  
    name,  
    city,  
    year  
  };  
};  
const person = createPerson("Alan Kay", "Springfield", 1940);  
console.log(person);
```

E se um **objeto** for retornado?

```
// Trocamos function pela arrow =>  
// porém, as chaves do objeto são interpretadas como BLOCO :( erro
```

```
const createPerson = (name, city, year) => {name, city, year};  
const person = createPerson("Alan Kay", "Springfield", 1940);  
console.log(person);
```

```
// Para funcionar, basta colocar parentes entorno do objeto  
const createPerson = (name, city, year) => ({name, city, year});  
const person = createPerson("Alan Kay", "Springfield", 1940);  
console.log(person);
```

InstanceOf

Com o operador `instanceof` é possível verificar se um objeto foi criado por meio de uma determinada função construtora analisando a sua cadeia de protótipos

```
// É possível verificar se um objeto foi criado  
// por meio de uma determinada função construtora  
// Analisando a cadeia de protótipo
```

```
const date = new Date();  
console.log(date instanceof Date);  
console.log(date instanceof Object); //descende de object  
console.log(date instanceof Array); // Na cadeia de protótipo não tem array
```

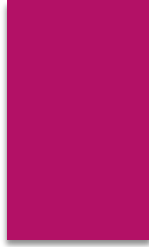
Qual é a diferença entre o **typeof** e o **instanceof**?

```
// typeof revela o TIPO DE DADO
```

```
const date = new Date();  
console.log(date instanceof Date);  
console.log(date instanceof Object);  
console.log(date instanceof Array);  
console.log(typeof date);  
// é OBJECT(tipo de dado) apesar de ser instancia de Date
```

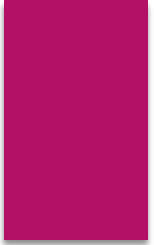
Tratamento de Exceções

Na linguagem JavaScript, **qualquer tipo de dado pode ser lançado como um erro interrompendo o fluxo de execução**



```
// função retângulo
const Rectangle = function(x, y) {
  this.x = x;
  this.y = y;
  this.calculateArea = function() {
    if (this.x > 0 && this.y > 0) {
      return this.x * this.y;
    } else {
      throw "Invalid value for x or y";
    }
  }
};

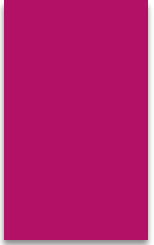
const rectangle = new Rectangle(10, 2);
console.log(rectangle.calculateArea());
```



```
// Vamos fazer um tratamento em CalculateArea
// Só calcula se x e y for MAIOR que 0
// Senão (throw) é invalido - INTERROMPIDO
```

```
const Rectangle = function(x, y) {
  this.x = x;
  this.y = y;
  this.calculateArea = function() {
    if (this.x > 0 && this.y > 0) {
      return this.x * this.y;
    } else {
      throw "Invalid value for x or y";
    }
  }
};

const rectangle = new Rectangle(-10, -2);
// forçando o erro com valores negativos
console.log(rectangle.calculateArea());
```

```
// PRECISAMOS FAZER UM TRY CATCH pra melhorar
// Vamos capturar o erro e exibir de maneira adequada
```

```
const Rectangle = function(x, y) {
  this.x = x;
  this.y = y;
  this.calculateArea = function() {
    if (this.x > 0 && this.y > 0) {
      return this.x * this.y;
    } else {
      throw "Invalid value for x or y";
    }
  }
};

try {
  const rectangle = new Rectangle(-10, -2);
  console.log(rectangle.calculateArea());
} catch (e) {
  alert(e);
}
```

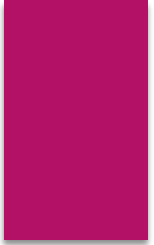
Destructuring

Por meio do **destructuring** podemos extrair valores de arrays e objetos de uma forma mais simples e direta

É possível **extrair os valores de um array**
criando variáveis em ordem, de acordo
com a posição de cada elemento

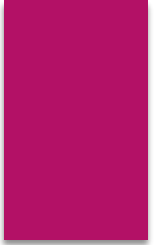
```
// Recurso que veio com o ES6 e trouxe flexibilidade para a linguagem
const language = "C;Dennis Ritchie;1972".split(";");
const name = language[0];
const author = language[1];
const year = language[2];
console.log(name, author, year);
```

```
// Vamos extrair os dados com Destructuring
// aqui todos os elementos
const language = "C;Dennis Ritchie;1972".split(";");
const [name, author, year] = language;
console.log(name, author, year);
```



Podemos **ignorar um elemento do array**
deixando de estabelecer um nome
para a variável

```
// aqui 2 elementos, repare na virgula  
const [, author, year] = "C;Dennis Ritchie;1972".split(";");  
console.log(author, year);
```



Assim como nas funções, é possível definir **valores padrão** para cada uma das variáveis

```
// Valores padrão como nas funções
const language = "C;Dennis Ritchie;1972".split(";");
const [name = "-", author = "-", year = "-"] = language;
console.log(name, author, year);
// com todos elementos não aparece
```

```
// tirando elementos
const language = "C;Dennis Ritchie".split(";");
const [name = "-", author = "-", year = "-"] = language;
console.log(name, author, year);
```

Para extrair os valores de um objeto é necessário referenciar a chave de cada uma das propriedades

```
// extrair valores de objetos
// forma direta, normal
const language = {
  name: "C",
  author: "Dennis Ritchie",
  year: 1972
};
const name = language.name;
const author = language.author;
const year = language.year;
console.log(name, author, year);
```

```
// extrair valores de objetos//
aplicando destructuring
const language = {
  name: "C",
  author: "Dennis Ritchie",
  year: 1972
};
const {name, author, year} = language;
//use chaves não colchetes em objetos
console.log(name, author, year);
```

É possível definir **nomes diferentes para as variáveis** em relação as chaves das propriedades do objeto

```
// Definir nomes diferentes para as variáveis  
// em relação as chaves das propriedades do objeto
```

```
const language = {  
  name: "C",  
  author: "Dennis Ritchie",  
  year: 1972  
};  
const {name: n, author: a, year: y} = language;  
console.log(n, a, y);
```

Também podemos referenciar as propriedades de objetos que estão dentro de outros objetos

```
// referenciar as propriedades de objetos  
// que estão dentro de outros objetos  
// É complexo, cuidado, as vezes é preferível criar as variáveis
```

```
const language = {  
  name: "C",  
  author: "Dennis Ritchie",  
  year: 1972,  
  company: {  
    name: "Bell Labs"  
  }  
};  
  
const {name: n, author: a, year: y, company: {name: c}} = language;  
console.log(n, a, y, c);
```


Podemos aplicar destructuring também nos **parâmetros de uma função**, tanto com arrays quanto com objetos

```
// Uma função normal
function sum(a, b) {
    return a + b;
}
console.log(sum(2, 2));
```

```
// destructuring - passar os parâmetros
// dentro de um array, e ao receber
// na função, restaurar como variáveis
```

```
function sum([a, b]) {
    return a + b;
}
console.log(sum([2, 2]));
```

```
// destructuring
// Vamos passar um objeto
// e restaurar como variáveis
```

```
function sum({a, b}) {
    return a + b;
}
console.log(sum({a: 2, b: 2}));
```

JSON

JSON, ou JavaScript Object Notation, é um formato de intercâmbio de dados, derivado da linguagem JavaScript que foi descoberto por *Douglas Crockford* e padronizado pela ECMA-404

[Pdf](#) [Livro](#) [Site do Douglas](#)



JSON

- Dominando o Mercado – rival do XML;
- Carrega arquivos de configuração;
- Na comunidade JS, quase tudo é JSON;
- A maioria dos Web Services utiliza para troca de dados;
- Existem algumas limitações no subset de dados aceito;
- Métodos stringify e parse

Pra que serve o JSON?



Tipos de Dados
Number
String
Boolean
Object
Array
null

Troca de dados entre sistemas!

JSON.stringify converte um determinado tipo de dado para JSON

```
JSON.stringify(10); // number - saida uma string
```

```
JSON.stringify("JavaScript"); // ? saida uma string com aspas
```

```
JSON.stringify(true); // boolean - similar ao number
```

```
JSON.stringify(false);
```

```
JSON.stringify({name: "Self", paradigm: "OO"}); // string contendo  
objeto com chaves e valores entre aspas
```

```
JSON.stringify([1,2,3,4,5,6,7,8,9]); // saida uma string
```

```
JSON.stringify(null); // similar ao number
```

O método **JSON.parse** converte um JSON para um determinado tipo de dado

```
JSON.parse('10');
```

```
JSON.parse('"JavaScript"');
```

```
JSON.parse('true');
```

```
JSON.parse('false');
```

```
JSON.parse('{"name": "Self", "paradigm": "OO"}');
```

```
JSON.parse('[1,2,3,4,5,6,7,8,9]');
```

```
JSON.parse('null');
```

Formas de comparar objeto com `stringfy`, formato de intercâmbio de dados.

```
const book1 = {  
  name: "Refactoring",  
  author: "Martin Fowler"  
};  
const book2 = {  
  name: "Refactoring",  
  author: "Martin Fowler"  
};
```

```
console.log(book1 === book2); // false
```

```
console.log(JSON.stringify(book1) === JSON.stringify(book2));  
// true
```

Clonar objetos (duplicar) com stringfy e parse

```
const book1 = {
  name: "Refactoring",
  author: "Martin Fowler"
};

const book2 = {
  name: "Refactoring",
  author: "Martin Fowler"
};

const book3 = JSON.parse(JSON.stringify(book2));

console.log(book2 === book3);

console.log(book3);
```